**MicroServices Notes**

**How To right size and identify services boundaries of mircroservices**

**Event Storming** is a collaborative workshop technique used to model complex systems and identify microservice boundaries. For example, in an e-commerce platform, stakeholders might gather to map out events like "Order Placed," "Payment Processed," and "Inventory Updated."

During the session, participants can visualize how these events interact and identify aggregates such as "Order," "Payment," and "Inventory."

**Key Steps in Event Storming for Microservice Sizing:**

**Identify Domain Events:**

Gather stakeholders to brainstorm and list all significant events in the system.

Example events: "User Registered," "Order Shipped," "Payment Failed."

**Group Events into Bounded Contexts:**

Organize events into clusters that represent distinct business capabilities.

For instance, events related to order processing can be grouped together, forming a bounded context for the "Order Management" microservice.

**Define Aggregates:**

Identify aggregates that encapsulate the state and behavior related to the events.

In the e-commerce example, the "Order" aggregate would handle events like "Order Placed" and "Order Canceled."

**Establish Boundaries:**

Draw boundaries around each bounded context to define where one microservice ends and another begins.

This helps in minimizing dependencies and ensuring that each microservice can operate independently.

**Iterate and Refine:**

As the system evolves, revisit the event storming sessions to adjust the boundaries and aggregates based on new insights or changes in business requirements.

**Example Application: E-commerce Platform**

**Events Identified**:

* **"Order Placed"**
* **"Payment Processed"**
* **"Inventory Updated"**

**Bounded Contexts:**

**Order Management**: Handles all events related to order processing.

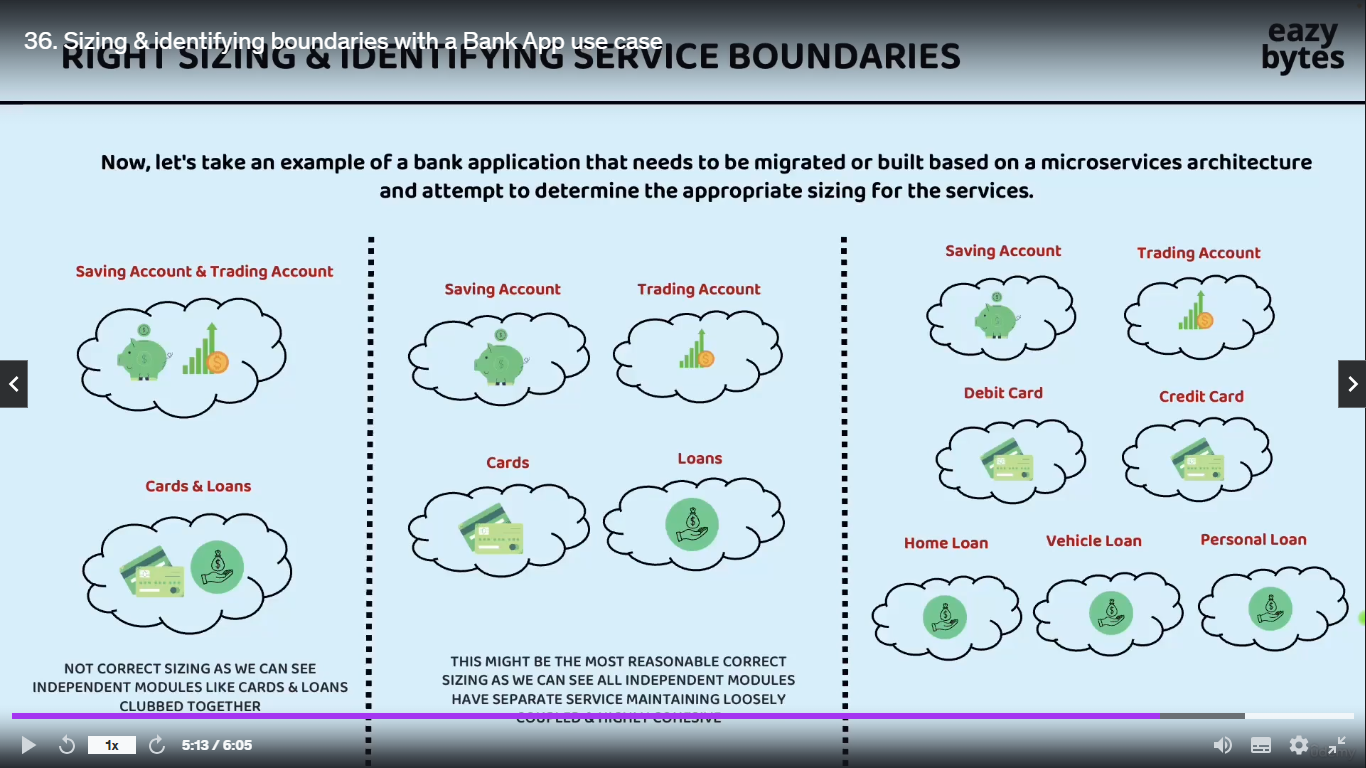
**Payment Processing**: Manages payment-related events.

**Inventory Management**: Responsible for inventory updates and stock management.

**Microservice Boundaries:**

Each bounded context corresponds to a microservice, allowing teams to work independently on their respective areas without affecting others.

This structured approach not only helps in sizing microservices effectively but also enhances collaboration among teams, leading to a more agile development process.

Now Sizing and identifying boundaries with a Bank App use case  
  


**Team Two**

**Team One**

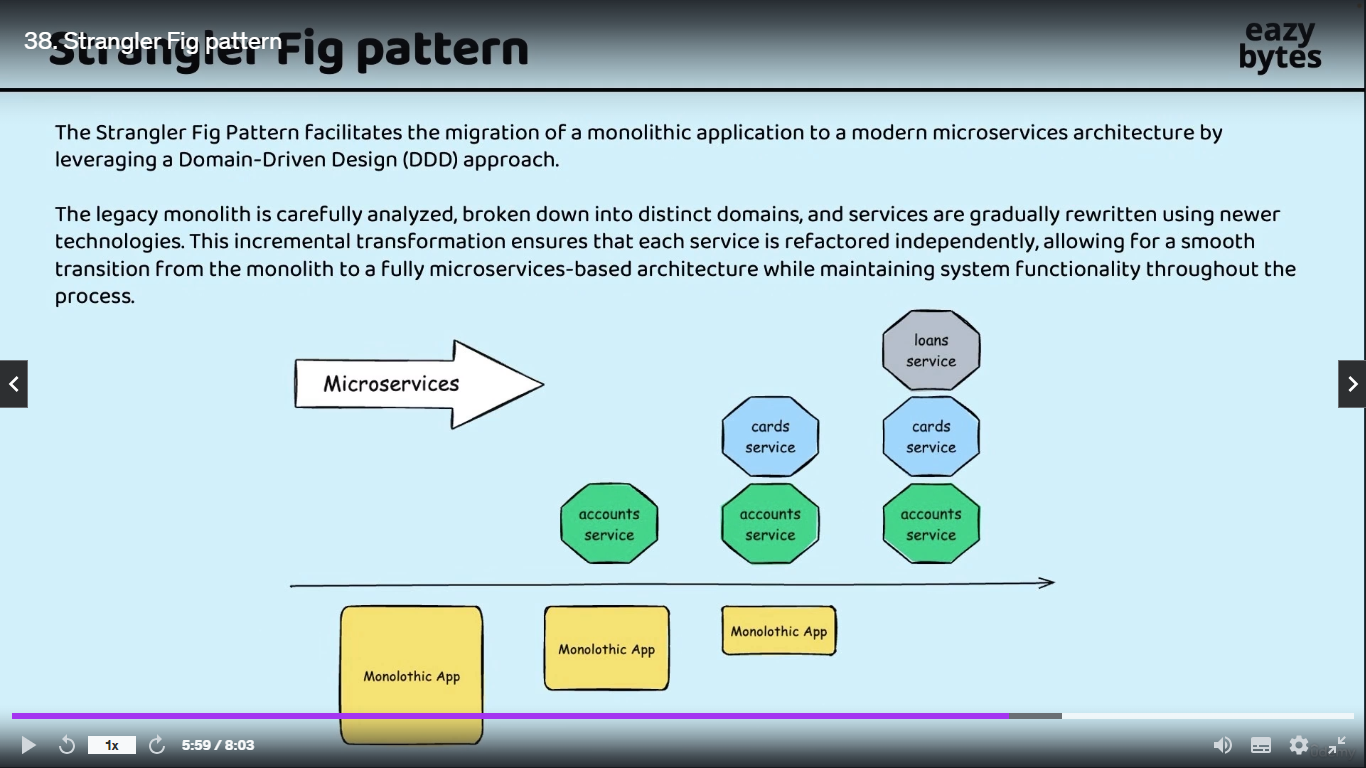
**Team Three**

**Coupled And Highly Cohesive**

Overall summary is no sizing Is right sizing initially and if there r some issues we should continously follow sizing your microservices till we reach to safer and correct sizing infact the expectation also is not identify the right sizing on the day one itself company or organization they will have their own learning with their own microservices sizing based on their learing they always try to right size and identifying service boundaries for their microservices.   
for now team two is winner as a cto/ceo to identify sizing and boundaries

**Strangle Fig Pattern** :- The Strangler Fig Pattern is a software migration strategy where a legacy system is gradually replaced by a new system by incrementally building new features alongside the old system and routing requests to the new components, allowing for a smooth transition without a complete rewrite at once.  
**When to use the Strangler Fig Pattern: -**

* **When** you need to modernize a large or complex legacy system
* When you want to avoid the risk associated with a complete system rewrite or “big bang” migration
* When the legacy system needs to remain operational during the transition to the new system

**  
What are Containers & How they are different from VMs(EasyByte Notes)**

**Create jar file from the springboot microservices**

1. Add this line in pm.xml file :- <**packaging>jar</packaging>**

<groupId>com.ms</groupId>  
<artifactId>accounts</artifactId>  
<version>0.0.1-SNAPSHOT</version>  
<packaging>jar</packaging>

1. Delete all the files of target folder
2. Check maven install or not :- mvn –version if not then should add in system path directory like java
3. Open microservices folder in cmd and run below command ,

Example: - **account> mvn clean install** to compile springboot application account is our springboot application now we can see in our target path of account microservice jar file of account microservice created.

1. Now using maven cmd we can run this jar file   
   cmd :- **mvn spring-boot:run**
2. Run jar using java command   
   **Command :- java –jar target/accounts-0.0.1-SNAPSHOT.jar**

**Create a docker file of account microservice**

1. Create a docker file inside the accounts microservice . Right click on account microservice select new file and Dockerfile (bcoz they don’t have any extention)
2. Write below command

#start with a base image containing java runtime  
From openjdk:17-jdk-slim  
  
#information around who maintains the image  
LABEL maintainer="MsBoss.com"  
  
#Add the Application's jar to the image   
Copy target/accounts-0.0.1-SNAPSHOT.jar accounts-0.0.1-SNAPSHOT.jar  
  
#execute the application  
ENTRYPOINT ["java","-jar","/accounts-0.0.1-SNAPSHOT.jar"]

3. Now our docker file is ready . now we will create the docker image of our account mircroservice using docker server by following below instruction

1. Firstly run docker bulid command   
   **docker build . -t boss215/accounts:s4**:- means we are telling to docker create the docker image using dockerFile which is present in account folder with the name Boss215/account:s4 , where Boss215 our docker user name and s4 is tag. We are execution this command from account folder so no need to specify the location so we are only using
2. Our image is created or not we can see using the below command  
   **docker images** :- below we can see info about newly created image  
   . **boss215/accounts s4 e3e3787b9659 2 minutes ago 167MB**
3. **I**f we are try to inspect this image we need to use below cmd  
   **docker image e3e7(image id first few char) :-**
4. **Now**  we are ready to convert this docker image to the docker container**Docker run** (using this we can create any number of container from dokcer image) **-p 8080:8080**(using this we need to provide port mapping bcoz by default all the docker container they are going to start insider their on isolated network and we can not access the services inside our docker n/w bcoz since it is deployed inside own private n/w so that’s y not them to access from the external n/w like from our local sys or any other sys so we need expose them explicitly with the help of this port mapping, that’s y we r giving the port first 8080 means expose the container at 8080 and second 8080 means container will running in 8080 inside private n/w) **boss215/accounts:s4**(This is our docker image from this we r trying to convet img to container)  
   **Final Cmd :- docker run -p 8080:8080 boss215/accounts:s4**

**Now** we can see our accounts microservices is running at port 8080 but we are not able to run any other cmd in same console so overcome this inconvience we will start the container in detach mode in detach mode(-d)   
Cmd :- **docker run -d -p 8080:8080 boss215/accounts:s4  
now we can run any other cmd inside our terminal  
7. Using** this command we can see how many container is in running mode   
**docker ps**

**Output :-**

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

6d02363828c2 boss215/accounts:s4 "java -jar /accounts…" 55 seconds ago Up 50 seconds 0.0.0.0:8080->8080/tcp nice\_tu

1. **docker ps –a :** using this we can see all container lists
2. **docker start 6d02363828c24bb4adc3bf8bc1f3d828176aa3142bd8c7(container id )**using this cmd we can run already created container
3. **docker stop 6d02363828c24bb4adc3bf8bc1f3d828176aa3142bd8c** to stop the running container

**Disadvantage of creating container using docker file**

Dockerfiles can become complex and difficult to maintain, especially as applications grow in size and dependencies.

In previous example we are using only four command but in future if we are using 100 of micrservices and command then it will be difficult to remember the all the command so , we will use the approach where dockerfile will created automatically without writing any low level instruction inside docker file .We have solution like  
**buildpacks** and **google jib**

**Generate Docker image of Loans microservice with Buildpacks**

Using Buildpacks we can transform our application source code into docker image that can run on any cloud there is no need to writing low level instruction with the help of docker file with a single maven command we can generate a docker image very easily.

**Step1**:- add these lines inside the maven dependency in pom.xml

<image>  
 <name>msbank/${project.artifactId}:s4</name>  
</image>

**${project.artifactId :- this means,it is getting name from project artifact**

**Step2** :- run this command :- **mvn spring-boot:build-image**

Based upon all the details and dependencies that we have mentioned in pom.xml is going to scan all the dependencies and files is going to generate the docker image

The command **mvn spring-boot:build-image** is used in the context of a Spring Boot application to build a Docker image of the application using the Spring Boot Maven plugin. This command simplifies the process of creating a Docker image by automatically configuring the necessary settings based on your Spring Boot application.

**Breakdown of the Command**

**mvn**: This is the command-line interface for Maven, a build automation tool used primarily for Java projects. It manages project dependencies, builds, and other project-related tasks.

**spring-boot**: This specifies that you are using the Spring Boot Maven plugin. This plugin provides various goals for building and managing Spring Boot applications.

**build-image**: This is a specific goal of the Spring Boot Maven plugin. It is responsible for building a Docker image of your Spring Boot application.

**Behind the Scenes**

When you run mvn spring-boot:build-image, several actions take place behind the scenes:

**Dependency Resolution**: Maven resolves all the dependencies specified in your pom.xml file. This includes Spring Boot dependencies and any other libraries your application needs.

**Application Packaging**: The Spring Boot application is packaged into a JAR or WAR file. This is usually done using the mvn package phase, which is part of the build lifecycle. The resulting artifact is typically located in the target directory.

Docker Image Creation:

**Base Image Selection**: The Spring Boot Maven plugin selects a suitable base image for your application. By default, it uses a minimal image that includes the necessary components to run a Spring Boot application.

**Configuration**: The plugin configures the Docker image based on the application properties and dependencies. This includes setting environment variables, exposing ports, and defining the entry point for the application.

**Copying Files**: The built JAR/WAR file is copied into the Docker image, along with any other necessary files (like configuration files).

**Dockerfile Generation**: The plugin generates a Dockerfile dynamically based on the configuration in your pom.xml and the properties of your application. This Dockerfile specifies how to build the image.

**Building the Image**: The generated Dockerfile is then used to build the Docker image using the Docker Engine. This process involves running Docker commands to create a new image layer by layer.

**Tagging the Image**: The resulting Docker image is tagged according to the specifications in your pom.xml or the command-line options provided (like -Dspring-boot.build-image.imageName).

**Example pom.xml Configuration**

To use the Spring Boot Maven plugin, you typically have a configuration section in your **pom.xml** like this:

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

<version>3.4.0</version>

<configuration>

<image>

<name>msbank/loans:s4</name>

</image>

</configuration>

</plugin>

</plugins>

</build>

**Conclusion**

The mvn spring-boot:build-image command streamlines the process of creating a Docker image for a Spring Boot application by handling dependency resolution, application packaging, Dockerfile generation, and image building. This automation allows developers to focus on writing code rather than managing the complexities of Docker image creation.

**Step 3** :- We will check our docker image is created or not

**Docker images**:- using this cmd we can see our image is created or not in our case our image is created with the name of loans

**Step 4**: Now we will generate the container from the loans image using below cmd

**docker run -d -p 8090:8090 msbank/loans:s4**

we can see at docker dashboard .our container is running on port 8090

“A **buildpack** is a set of scripts and tools that automate the process of transforming application source code into a runnable container image by providing the necessary dependencies, configuration, and runtime environment. Buildpacks are commonly used in platforms like Cloud Foundry and Heroku, as well as in the Spring Boot Maven plugin for building Docker images”

**Create a docker container using jib(jib only used for java application only )**

**Step1** :- add packaging details after versioning in pom.xml

<version>0.0.1-SNAPSHOT</version>  
<packaging>jar</packaging>  
<name>cards</name>

And add the new dependencies under the build in pom.xml ….shown below

<plugin>  
 <groupId>com.google.cloud.tools</groupId>  
 <artifactId>jib-maven-plugin</artifactId>  
 <version>3.3.2</version> *<!-- Check for the latest version -->* <configuration>  
 <to>  
 <image>msbank/${project.artifactId}:s4</image>  
 </to>  
 </configuration>  
</plugin>

**Step 2**:- run the command to generate the docker image   
**mvn compile jib:dockerBuild** :- if we try to run this cmd it will scan all the details inside our pm.xml and it is going to generated docker image for our card microservices this is faster than **Buildpacks**

**We can see the our card docker image is created**

msbank/cards s4 af26c0fd6255 54 years ago 325MB

**Step 3**: docker run –d –p 9090:9090 msbank/cards:s4  
now our container is started

Now we can try to hit carts api using post man and we can see our application is running at 9090 port properly ……**BOOM**

**We can directly created the docker image and push into docker hub or cloud(gcp , aws) without installing the docker in our local sys . but we have to need give the credentials and change some line in our bulid dependencies**

<plugin>  
 <groupId>com.google.cloud.tools</groupId>  
 <artifactId>jib-maven-plugin</artifactId>  
 <version>3.3.2</version> *<!-- Check for the latest version -->* <configuration>  
 <to>  
 <image> your-dockerhub-username/your-app-name </image>

Or

<image> gcr.io/PROJECT\_ID/IMAGE\_NAME</image>

Or

<image> your-aws-account-id.dkr.ecr.REGION.amazonaws.com/IMAGE\_NAMEimage>  
 </to>  
 </configuration>  
</plugin>

Run this command to create the image :- **mvn compile jib:build**

**Pushing Docker images from local to remote docker hub repository**

**Cmd to push the img to docker hub**:- **docker image push docker.io/boss215/accounts:s4**

Now we can see at dashboard of docker our project successfully upload/push to the docker hub :- click on images -> hub , inside hub we can see the our images successfully pushed into the docker hub

**Note :-** if we are not using our username of docker hub as a project name at the time of creating the image then we will not able to upload/push image to docker hub so , we are using username before the image name at the time of creating the docker image :- which is **boss215**

**Using below command we can pull the image from docker hub**

**docker push boss215/accounts:s4**

**Docker Compose**

Previously we have created the three docker images of our microservices if we want to start these microservices/images we have to write cmd three time to run all three container. writing the cmd for every images is very time consuming if we have lots of image .To overcome this problem docker provide Docker Compose

**Docker Compose** is a tool for defining and running multi-container Docker applications using a simple YAML configuration file

**Real-Time Example**: For a web application that consists of a front-end service (e.g., a React app), a back-end service (e.g., a Node.js API), and a database service (e.g., PostgreSQL), you can define all three services in a **docker-compose.yml** file. When you run **docker-compose up**, it will automatically start all the containers, set up their networking, and ensure they can communicate with each other seamlessly.

Step 1 : Check docker compose is install or not docker compose version

Step 2: -create a configuration file we can create anywhere in our project but we r going to create in accounts microservice. Create a new file in accounts microservice with the format. .yml bcoz we r going to provide all our project configuration inside this … our file is

**docker-compose.yml**

services:  
 accounts:  
 image: "boss215/accounts:s4"  
 container\_name: accounts-ms  
 ports:  
 - "8080:8080"  
 deploy:  
 resources:  
 limits:  
 memory: 700m  
 networks:  
 - msbank  
 loans:  
 image: "boss215/loans:s4"  
 container\_name: loans-ms  
 ports:  
 - "8090:8090"  
 deploy:  
 resources:  
 limits:  
 memory: 700m  
 networks:  
 - msbank  
  
 cards:  
 image: "boss215/cards:s4"  
 container\_name: cards-ms  
 ports:  
 - "9090:9090"  
 deploy:  
 resources:  
 limits:  
 memory: 700m  
 networks:  
 - msbank  
  
networks:  
 msbank:  
 driver: "bridge"

Explanation :-

This YAML file is a Docker Compose configuration that describes a multi-container application with three services: `accounts`, `loans`, and `cards`. Each service corresponds to a different microservice in a banking application and is organized under the `services` key. Below is a breakdown of each section:

### Services

Each service is defined under the `services` key and contains various settings:

**1. Accounts Service:**

**imag**e: Specifies the Docker image to use for this service. In this case, it's `boss215/accounts:s4`, where `s4` denotes a specific version or tag of the image.

**container\_name**: This sets a specific name for the running container, which is `accounts-ms` here.

**ports**: Maps port `8080` on the host to port `8080` on the container, allowing external access to the service via the mapped port.

**deploy**: Indicates deployment configuration, specifically resource limits.

**resources**: It specifies the limits for the resources the container can use.

**limits**: In this case, it restricts the memory usage to 700 MB for the service.

**networks**: Connects the service to the `msbank` network.

**2. \*\*Loans Service:\*\***

- Similar to the accounts service, this service uses the image `boss215/loans:s4` and has the container name `loans-ms`.

- Exposes port `8090` on the host to port `8090` in the container.

- Has the same memory limit configuration (700 MB) under the `deploy` key.

- Is also connected to the `msbank` network.

**3. Cards Service:**

- Again, similar to the previous services, this one utilizes the image `boss215/cards:s4` and has the container name `cards-ms`.

- Exposes port `9090` on the host to port `9090` in the container.

- Includes the same memory limit (700 MB) in the `deploy` section.

- Connects to the `msbank` network.

**### Networks**

**msbank**: This section defines a custom network named `msbank`.

**driver**: Specifies the network driver to use, in this case, it is set to `bridge`, which creates a private internal network for containers to communicate with each other while being isolated from the host network.

**### Summary**

In summary, this Docker Compose file sets up three microservices (`accounts`, `loans`, and `cards`) for a banking application. Each service has its own container configuration, resource limits, and network settings. They can communicate with each other through the custom `msbank` network. The specified ports allow external access to these services. The memory limits help to ensure that the services do not consume more than the specified amount of memory, which can be crucial for maintaining performance and resource management in a production environment.  
**NOTE :- Using the same network for all microservices allows seamless intercommunication, enabling them to call each other efficiently and effectively share data within our banking applicatio**and effectively share data within our banking application.

**Step 2 :-** Now .we will Run All the microservices containers using Compose command. Please make sure this docker cmd should run from the location where our **.yml** file is located. Our file is located at accounts.

E:\Microservices\section2\accounts> **docker compose up –d**  
**using** this single cmd our all the microservices is running we can see in docker container as well as in console ….Boom

**Step 3**:- To delete all the container we can use below   
**Delete cmd**:- **docker compose down ……**This cmd will delete the container which is best practice …..but we don’t want to deltete the container just want to stop the container then we will use the below cmd **Stop Cmd :-docker compose stop**

**Docker Extention and logs Explorer:- we** can add and downloads the extention which in docker after click on add extension link . we are downloading the log Explorer using this we can see all the logs related to our containers . According to need we can add any extentions which will make easy our work



****

**Native Application**

**What are cloud native application** :- A cloud-native application is a software application designed to take full advantage of cloud computing architectures, leveraging microservices, containerization, continuous integration and deployment (CI/CD), and scalability to deliver enhanced performance, flexibility, and resilience.

“**A cloud-native application is an application that is built specifically to run in cloud environments, utilizing cloud services and technologies to achieve scalability, resilience, and flexibility**.”

**Real-Time Example**

**Netflix**: Utilizes a cloud-native architecture to deliver streaming services to millions of users globally. This allows Netflix to scale its services dynamically based on user demand, ensuring high availability and quick updates without downtime.

**Characteristics of cloud Native App**

Cloud-native applications possess several key characteristics that enable them to thrive in cloud environments. Here are some of the main attributes along with real-time examples:

**Microservices Architecture**: These applications are built as a collection of loosely coupled services, allowing for independent deployment and scaling.

\***Example** : Spotify uses microservices to manage different functionalities like music streaming, user accounts, and playlists, enabling rapid updates and scalability.

**Containerization**: Cloud-native applications are often packaged in containers, which provide a lightweight and portable environment for running applications.

Example: Docker is widely used by companies like Airbnb to deploy their applications in containers, ensuring consistency across different environments.

**Dynamic Scaling**: These applications can automatically scale up or down based on demand, optimizing resource usage and cost.

**Example**: Amazon Web Services (AWS) allows applications like Snapchat to scale dynamically during peak usage times, such as during major events.

**DevOps Practices**: Cloud-native applications are developed and deployed using DevOps methodologies, promoting collaboration between development and operations teams for faster delivery.

**Example**: Etsy employs DevOps practices to continuously integrate and deploy new features, enhancing user experience and reducing time to market.

**API-Driven Development:** They utilize APIs for communication between services, enabling flexibility and integration with other applications.

**Example**: Twitter provides APIs that allow third-party developers to build applications that interact with its platform, enhancing functionality and user engagement.

**Resilience and Fault Tolerance**: Cloud-native applications are designed to handle failures gracefully, ensuring high availability.

**Example**: Google Cloud services are built with resilience in mind, allowing applications like YouTube to remain operational even during server outages.

**Infrastructure as Code (IaC):** This approach allows for automated management and provisioning of infrastructure, leading to consistent environments.

**Example**: Netflix uses IaC tools like Terraform to manage its cloud infrastructure, enabling rapid deployment and scaling of services.

These characteristics collectively enable cloud-native applications to be agile, efficient, and responsive to changing business needs, making them ideal for modern software development.

**Cloud Native vs. Traditional Enterprises App**

**Architecture**

**Cloud-Native Applications**: Built using microservices architecture, where applications are composed of small, independent services that can be developed, deployed, and scaled independently.

**Traditional Enterprise Applications**: Typically monolithic, meaning they are built as a single, unified unit. Changes or updates often require redeploying the entire application.

2. **Deployment and Scalability**

**Cloud-Native Applications**: Designed for dynamic scaling and can automatically adjust resources based on demand. They are often deployed in containers, allowing for quick and efficient scaling.

**Traditional Enterprise Applications**: Scaling often requires significant manual intervention and can involve complex processes, such as adding more hardware or reconfiguring existing systems.

3. **Development Practices**

**Cloud-Native Applications**: Embrace DevOps practices, enabling continuous integration and continuous deployment (CI/CD). This allows for rapid iteration and faster time to market.

**Traditional Enterprise Applications**: Development cycles are typically longer, with more rigid processes and less frequent updates, often leading to slower response times to market changes.

4. **Infrastructure Management**

**Cloud-Native Applications**: Utilize Infrastructure as Code (IaC) for automated provisioning and management of infrastructure, leading to consistency and repeatability.

**Traditional Enterprise Applications**: Often rely on manual configuration and management of physical or virtual servers, which can lead to inconsistencies and increased operational overhead.

5. Resilience and Fault Tolerance

**Cloud-Native Applications**: Built with resilience in mind, often incorporating features like automatic failover and self-healing capabilities to ensure high availability.

**Traditional Enterprise Applications**: May not be designed for fault tolerance, leading to potential downtime during failures or maintenance.

6. **Cost Structure**

**Cloud-Native Applications**: Typically follow a pay-as-you-go model, allowing organizations to pay only for the resources they use, which can lead to cost savings.

**Traditional Enterprise Applications**: Often involve significant upfront capital expenditures for hardware and software licenses, along with ongoing maintenance costs.

**7. User Experience and Accessibility**

**Cloud-Native Applications**: Designed for accessibility from anywhere, often providing a better user experience through responsive design and cloud-based features.

**Traditional Enterprise Applications**: May be limited to specific environments (e.g., on-premises) and can be less user-friendly, often requiring specific hardware or software configurations.

**Real-World Examples**

**Cloud-Native Application:** Netflix is a prime example of a cloud-native application, utilizing microservices and cloud infrastructure to deliver streaming services efficiently and at scale.

**Traditional Enterprise Application**: SAP ERP systems are often considered traditional enterprise applications, typically deployed on-premises and requiring significant resources for maintenance and updates.

In summary, cloud-native applications are designed for the cloud environment, emphasizing agility, scalability, and resilience, while traditional enterprise applications are often more rigid, monolithic, and resource-intensive.

15 factor methodology

**Configuration Management in Microservices**

**Configuration Challenge** :- The challenge may we faced while building microservice this challenge is about configuration management inside the microservices

**C1** :- inside microservices how do we separate the configuration/properties from the our business logic bcoz without separating the configuration/properties from our microservices business logic we can not reuse the same docker image across multiple environment  
if we club all our business logic and configuration together then for each environment we need to create a separate decker image along with the relevant properties/configuration . this is not recommended approach to create docker image again and again for each environment . **make sure we will using the same docker image for all type of environments including the production with such requirement** …… how we will separate he configuration from the business logic this is very first question that we have

**C2** :- how do we inject configuration/properites at run time that are needed by our microservice during the start up of our microservices .Some of the sensitive properties like credentials we can not mention them inside any configuration or any inside any business logic such sensitive properties/configuration we need to make sure we are injecting them into microservice during the start up of micrservice…

**C3** :- In the very similar line whenever we r dealing with configuration/properties we need to make sure we are maintaining all this in a single   
In a centralize repository along with a versioning of them bcoz inside monolithic app we only one or two application and it is very easy to maintain all our configuration whenever we want .. whereas 100 of microservices it is going to be super complex to maintain all the properties of all the microservices manually that why when we try to build microservice we need to make sure we are maintaining all properties in the centralize repository along with versioning of them.

**we have all these chanllages in term of configuration management in microservices**

There is multiple solution available in springboot ecosystem to handle this challenge Below are the solutions . let’s try to identify one suites for microservices

* Configuration spring boot with properties and profile
* Applying external configuration with Spring boot
* Implementing a configuration server with spring cloud config server

**How configuration Handled in traditional APPs & Microservices**

**When** we are trying to build traditional app or monolithic app all our source code along with the configuration file they we will be bundled together and with that whenever there is a different configuration is needed for a different environment we need to rebuild the application code again with the required configuration so with this there is no guarantee that application would behave consistently boz the main buiseness logic may differ from one environment to other environment when we r trying to build the application this approach might have work for a monolithic application bcoz there is only one application and they can do multiple build based upon the environment whereas

with **microservice** since they will have 100s of microservice doing multiple build again and again for each environment with the required configuration data is going to be super2 complex that why as per the 15 factor methodology all the configuration which are going the change between deployment such as credentials service urls or resource handle all such configuration data we need to provide outside our build component so all such configuration that are likely changed we need to maintain oustside our business logic so that the application artefact will be immutable across all environment once we do build and generate a docker image for our microservice the same image we can use across environment and the configuration has to be injected from the external location this way the application build remain unchanged across all the environment

**How Configuration work in Spring boot**

Springboot lets us externalize our configuration so that we can work with the same application code in different2 environments. We don’t have to rebuild our application again again ther are different approaches that springboot follow to externalize the configuration sources and these approaches include with the help of properties file and yml file where we can define all the require configuaration nd properties that our app/microservice need during the startup of the service apart from this property/yml file we can also inject the configuration/propertied during the startup of the service with the help of env. Variable and command line argument

By default our spring boot application is going to look configuration/properties that we have mentioned inside the application.properties/application.yml file . apart from these default name we can also have other property file name and we can make our springboot to read from them during the start up of the application but if we try to mentioned the property/configuration inside our property file we are clubbing everything along with the source code so this approach may not work to great action that why springboot also support override the default value that we have defined inside the application.properties file may be for development environment we might have mentioned the default database credentials but at run time inside our QA or production deployment we want to override the default value like username, password, url, details of database during the startup of the application that s why to overcome this challenge we can always override the properties with the other approaches like command line argument environment variables so

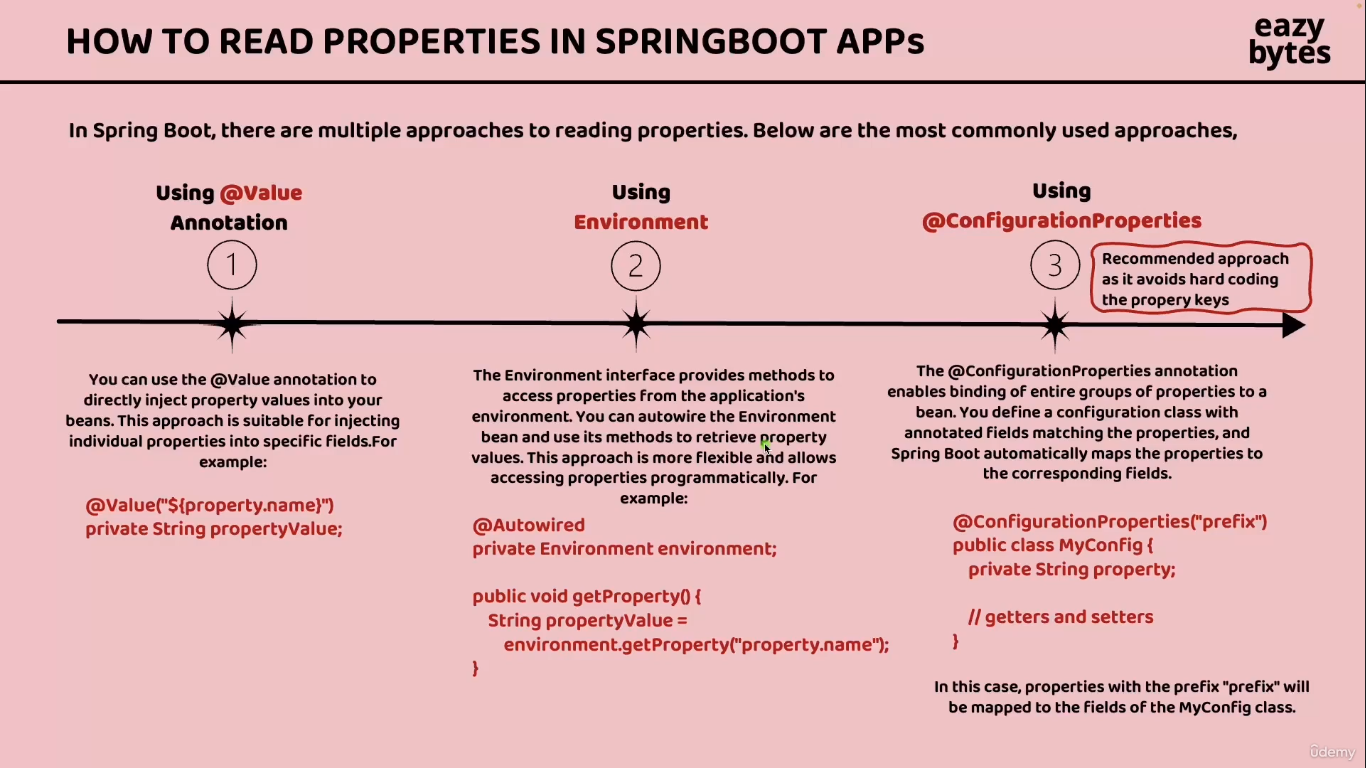
here may have a question like if I mentioned the same properties in multiple location will they we any preference or priority that springboot follows we can see below all the most commonly approaches to provide the configuration inside our springboot applications

**priority** :- lower items overriding earlier ones

* Application.properties
* OS Environmental variables
* Java sys properties (System.getProperties())
* JNDI attributes form java:comp/env
* ServletContext init parametes
* ServeletConfig init parameters
* Command line arguments

Whereas application.properties having lowest priority and command line argument having highest priority

**How to Read properties in springboot Apps**



**Reading Configuration using @Value Annotation**

Step 1:- add/define the string or something in application.yml  
Example :-

build:  
 version: "1.0"

Step 2 :- Fetch in Controller using @Value annotation

*@Value*("${build.version}")  
*private* String buildVersion;

Now build version value will store in buildversion variable and we can use this value

If we have 100s of microserices and have 100 different properties then r we going to create 100s of different filed inside our microservices that not a viable solution and same time in this approach for injecting the java field we have need to hard coded the property name like build.version ..creating a java filed and mentioning a hardcoded property value for every microservice is not going to be feasible option that’s why this approach Is only if we have only one or two property…

**Reading configuration using Environment Interface**

This approach is help us to read the environment properties defined inside the environment  
Where we have deploy our microservices.

**Why we have to need to define some properties as environment variable why cant we directly defined them inside the application.yml** .  
Bcoz some sensitive information like password or any other information we can not define them inside the applicaition.yml or any other place where it will expose our sensitive information with that reason it is always advisable to define the sensitive configuration details as a environment variable only so that no one can see those value they not have access to our production server only the server admin they will have access this way we can trying to secure our application whenever dealing with the sensitive information

This approach involves interface with the name environment So  
**Step1** :- autowired the environment interface inside controller

*@Autowired  
private* Environment environment;

**Step 2**:- call the properties using environment object like below

*@GetMapping*("/java-version")  
*public* ResponseEntity<String> getJavaVersion() {  
 *return* ResponseEntity.status(HttpStatus.OK)  
 .body(environment.getProperty("JAVA\_HOME"));  
 }

This environment is system environment : like we have set path at system environment from where it is fetching the java path...  
output :- **users/ms/skman/cndidaes/java/current** …this simply giving the JAVA\_HOME folder location

Using this approach we can only read the environment property not the properties which I have define inside our applications,yml .so This approach also have disadvantage like we can only read one property at a time and same time we need to hard code the property key name inside our java code this approach also going to work only we have one or two properties if we have higher number of environment properties that we want to read then definitely this approach is not recommended will see some advance options that we have in third approach

**Reading Configurations using @Configuration**

Using this approach we can read multiple properties at a time with a single POJO Class.  
All the limitation that we have with the previous two approaches will be handled with this third approach limitation are like where we hard coded the property key name inside the java code and the same time we can only read one property at a time with that two approaches that we have discussed so to demo to third approach we are going to create a set of properties that are required for our account mircrosevices wheneve we r trying to use   
**@Configurtion propert** approaches first we need to make sure all our properties have a common prefix name for the same we going to create a prefix name with the name accounts(inside this property we can define any number of property see in step 1) we can see in step one

**Step 1** :- write a proeprties in Appication.proeprties

accounts:  
 message: "Welcome to MsBank accounts related local APIs "  
 contactDetails:  
 name: "John Doe - Developer"  
 email: "john@msbank.com"  
 onCallSupport:  
 - (555) 555-1234  
 - (555) 523-1345

**Step 2** :- Create a POJO Record for the **accounts** properties which we have write in application.property

*@ConfigurationProperties*(prefix="accounts")  
*public record* AccountContactInfoDto(String message, Map<String,String> contactDetails, List<String> onCallSupport) {  
}

this POJO record map the accounts property with java record (we can not change the value of the record filed we can only fetch the data using the getter ) and configuraionproperties map the properties with this POJO

**Step 3** :- at the application class enable the configuration for this record/class using below

*@EnableConfigurationProperties*(value={AccountContactInfoDto.*class*})

**Step 4**:- Inject the POJO record and write a API for fetching contact details

*@Autowired  
private* AccountContactInfoDto accountsContactInfoDto;

*@Operation*(  
 summary = "Get Contact Info",  
 description = "Contact Info details that can be reached out in case of any issues"  
)  
*@ApiResponses*({  
 *@ApiResponse*(  
 responseCode = "200",  
 description = "HTTP Status OK"  
 ),  
 *@ApiResponse*(  
 responseCode = "500",  
 description = "HTTP Status Internal Server Error",  
 content = *@Content*(  
 schema = *@Schema*(implementation = ErrorResponseDto.*class*)  
 )  
 )  
}  
)  
*@GetMapping*("/contact-info")  
*public* ResponseEntity<AccountsContactInfoDto> getContactInfo() {  
 *return* ResponseEntity  
 .status(HttpStatus.OK)  
 .body(accountsContactInfoDto);  
}

Hit the url and get the output boom :- <http://localhost:8080/api/contact-info>

Output

{

    "message": "Welcome to MsBank accounts related local APIs ",

    "contactDetails": {

        "name": "John Doe - Developer",

        "email": "john@msbank.com"

    },

    "onCallSupport": [

        "(555) 555-1234",

        "(555) 523-1345"

    ]

}

If we have need to different version for different environment then we will use **profile** concept for that

**Profile**

The challenge with previous application.property file reading approach is if we try to deploy this code into various environment the set of property they r going to use will be same what if we have requirement where we should used different property value inside different2 environment we will be seen such requirement very often inside real project for example take database credentials itself . The properties related to database credentials they should not have the same value inside all the environment based upon the environment they should have different 2 properties so let see how to overcome this challenge and how springboot is going to help in this scenario inside springboot we have concept **Profile**

“**Spring provides a great tool for grouping configuration properties into so-called profiles(Dev ,QA, prod)”** which means we can create different set of file and properties that will get activated based upon the current execution environment if we create three different profile like dev, prod, Qa this will give flexibility to us to have different values for our configuration and the same will be activated based upon the current active profile on based upon the current executing env using this profile we can perfectly set our application that will run in different environment with the same code but are application is going to use different properties or configuration using the same profile also we can control the bean creation process we can write logic such a way that our bean has to be created only when a particular profile is active this way this profile concept inside springboot it can influence the application properties that will load and the bean that will get created inside the spring context

By default inside the springboot framework the default profile is always active whatever properties and configuration we have defined inside our application.propeties and applications.yml they will go into the default profile and these profile will always be activated by default and based upon our requirement we can create another profile by creating properties file or yml files by folooowing a naming conventions like we can see if we want to create two more profile for the production environment and QA environment we can create the files like applications.prod\_properties or applications\_qa.properties

In our case we will create file with the .yml extentions   
once we create these two extra profile files then inside our application they will be total three profiles  
one is default   
second is production  
third is QA  
This way we can create any number of profile based upon our business requirement

**How to activate an specific profile**

We can activate a specific profile using spring.profiles.acitve property like below

**Spring.profiles.active=prod**

**Demo of Springboot profiles inside accounts microservices**

**Step1** :- create the **application\_prod.yml** and **application\_qa.yml** inside the resources application now we have 3 application files (qa , prod, default(application.yml))

**Application.yml**

server:  
 port:8080  
  
spring:  
 datasource:  
 url: jdbc:h2:mem:testdb  
 driver-class-name: org.h2.Driver  
 username: sa  
 password: ''  
 h2:  
 console:  
 enabled: true  
 jpa:  
 database-platform: org.hibernate.dialect.H2Dialect  
 hibernate:  
 ddl-auto: update  
 show-sql: true  
  
  
  
build:  
 version: "1.0"  
  
accounts:  
 message: "Welcome to MsBank accounts related local APIs "  
 contactDetails:  
 name: "John Doe - Developer"  
 email: "john@msbank.com"  
 onCallSupport:  
 - (555) 555-1234  
 - (555) 523-1345

In the default properties file we are using database and port number as a same in all of the files but we will using different account properties (like message account details etc….)

|  |  |  |
| --- | --- | --- |
| **Application\_qa.yml** | **Application\_prod\_.yml** | **Application.yml** |
| spring:  config:  activate:  on-profile: "qa"  build:  version: "2.0"  accounts:  message: "Welcome to MsBank accounts related QA APIs "  contactDetails:  name: "Smitha Ray - QA Lead"  email: "smitha@msbank.com"  onCallSupport:  - (666) 265-3765  - (666) 734-8371 | spring:  config:  activate:  on-profile: "prod"  build:  version: "1.0"  accounts:  message: "Welcome to MsBank accounts related prod APIs "  contactDetails:  name: "Reine Aishwarya - Product Owner"  email: "aishwarya@msbank.com"  onCallSupport:  - (453) 392-4829  - (236) 203-0384 | server:  port: 8080 spring:  datasource:  url: jdbc:h2:mem:testdb  driverClassName: org.h2.Driver  username: sa  password: ''  h2:  console:  enabled: true  jpa:  database-platform: org.hibernate.dialect.H2Dialect  hibernate:  ddl-auto: update  show-sql: true  config:  import:  - "application\_qa.yml"  - "application\_prod.yml"  profiles:  active:  - "prod"   build:  version: "3.0"  accounts:  message: "Welcome to EazyBank accounts related local APIs "  contactDetails:  name: "John Doe - Developer"  email: "john@eazybank.com"  onCallSupport:  - (555) 555-1234  - (555) 523-1345 |

By default our application.properties file will be our property file we can change using below code which is also written in application.property file

config:  
 import:  
 - "application\_qa.yml"  
 - "application\_prod.yml"  
profiles:  
 active:  
 - "prod"

but if we want to change this **prod** to **qa** then we have to need to change in application property file and re-generate the docker image so over come this process we will change using from another value like …using **cmd** line **JVM** & environment options

**Externalization configuration using command line argument**

Springboot automatically converts command-line arguments into key/value pairs and adds them to the environment object. In a production application this becomes the property source with the highest precedence**. We can** customize the application configuration by specifying command line arguments when running this JAR you built earlierwe will use the below cmd to externalization configuration

**java –jar accounts-serivice=0.0.1-SNAPSHOT.jar --build.version=”1.1”**

The command-line argument follows the same naming convention as the corresponding spring property with the familiar **--prefix** for CLI argument

**How to externalized configuration using JVM system properties**

This jvm system property having less precendence as compare to command line argument but it has more precedence compare to the normal property files like application.yml file

**Java –Dbuild.version=”1.2” –jar aacounts-service-0.0.1-SNAPSHOT.jar**

if we have same proeperty in command line and jvm sys then precendence will comes to the picture . In the scenario where both a jvm and system property and a command line argument are specified the precendence rules dictate that spring will prioritize the value provide as a command line argument. This means that the value specified through the CLI will be utilize by the application taking precendece over the jvm proeprties

**How to externalized configuration using environment variables**

Environment variables are widely used for externalize configuration as they offer portability across different operating system as they are universally supported .Most programming languages including java provide mechanism to access environment variables such as the System.getevn() method.

We can access this env variable inside the java code with the help of Sytem.getenv()  
TO map a spring property key to an environment variable we need to convert all letters  
to uppercase and replace any dots or dashed with the underscores. Spring Boot will handle this mapping correctly internally. For example an environment variable name BUILD\_VERSION will be recognized as the property build.version. This feature is known as relaxed binding

For example if we want provide the property of build.version as an environment variable the we need to make sure we are following this upper standard .

**Windows  
env:BUILD\_VERSION=”1.3”; java –jar accounts-service-0.0.1-SNAPSHOT.jar**

**Linux based OS   
BUILD\_VERSION=”1.3”; java –jar accounts-service-0.0.1-SNAPSHOT.jar**

**Activating the profile using command line JVM & Environment options**

**1. Activating the profile using command line** :- right click on our application file from where our springboot application will start and click on modify Run configuration file and write this cmd in Program argument **--spring.profiles.active=qa --build.version=1.1**

**Output :-** activate the **qa** profile and changing the build version to 1.1

2. **Activating the profile using JVM System Variable** :- right click on application main file from where our app will start and click on modify Run configuration file and then click on modify options and then select Add VM options after click on that we can see one blank text box shown on the screen with the placholder VM options here write the same value and change one thing add D before the command shown below

**-Dspring.profiles.active=prod -Dbuild.version=2.0** make sure before add this remove the cmd line setting bcoz cmd line have more precedence as compare to jvm system and we can re run application after that we can see profile will change into prod

3. **Activating the profile using environment variable** :- right click on modify run configuration remove the jvm system setting which we r write for the jvm sys  
and write the this line/cmd in environment variable   
**SPRING\_PROFILES\_ACTIVE=prod;BUILD\_VERSION=3.0 :- now** prod profile activated and build version is 3.0

If we are writing cmd in all the three like in cmd line and sys variable and environment the command line approach will update the profile bcoz cmd line have higher precedence if

we have jvm sys and environment approach then jvm sys one updated bcoz jvm have high precedence as compare to environment approach

**Drawbacks of externalized configuration using Springboot alone**

**Complexity in Configuration Management**

**Challenge**: As applications grow and require numerous configurations (database, API endpoints, feature flags, etc.), managing these configurations can become complex. Multiple files for different environments can lead to confusion.

**Example**: An application with separate application-dev.yaml, application-test.yaml, and application-prod.yaml files can be challenging to manage, especially if the configurations differ greatly between environments. Developers may accidentally apply the wrong configuration file.

**Difficulty in Debugging**

**Challenge**: Debugging issues related to configuration can become difficult, particularly when configurations are loaded from multiple sources (properties files, environment variables, command-line arguments). Identifying where a config is set can take time.

**Example**: If a database connection fails in production, it might be due to a configuration value being set incorrectly in the environment variable. Tracing where that variable was set or if it was overridden by the command-line arguments can be cumbersome.

**Sensitive Data Exposure**

**Challenge**: Externalized configurations often include sensitive information (e.g., API keys, database passwords). If not managed properly, these can be exposed, especially in version control systems.

**Example**: A application.properties file containing database credentials might get accidentally committed to a public Git repository. This can lead to security vulnerabilities, including unauthorized access to the database.

**1.** We trying to externalize configuration using the approaches like CLI argument and JVM properties and environment variables they are the effective way to aexternialize the configuration and they also support to maintain the immutability of our application build we don’t have to regenerate the docker image again and again for different environment until this point everything is perfect now

We saw whenever we r trying to use this approach it often involves to executing separate command with the help of java command or manually setting up the application which means we need to inject the cli argument or jvm properties environment variable manually at some point of time may be inside the CI/CD pipeline so this process again may introduce potential errors during the deployment again we can see we need to dependent on someone like they should be some human setting up all these externalize configuration during the startup of the application and doing that for all instances for all the microservices is going to be super challenging we can automate most of this task with the help of CI/CD pipeline like github action or Jenkins but still there is chance that some one can meshup this process so that a very first limlitation that we have here

2. if we have 100s of microservice then we will also have 100s of configuration properties that will evolve and changes on day to day bases just like how our application code we should also follow the strategies to store and maintain the versioning of our configuration and proeprties based upon a release and same time we should also support the auditing functionaliy like who access our configuration which client access our configuration data with the springboot profle alone we are putting all our configuarion inside the source code itself so anyone who has access to our source code or docker image they can easily understand all our configuration and it is not a vice decision to expose the the configuration to any one that why it is always recommended to store our configuration seperatly inside a centeralzed repository where it support all kind of versioning tracking reviosion of auditing the configuration right now springboot profile can not profile this configuration that’s why we can this as limitation and

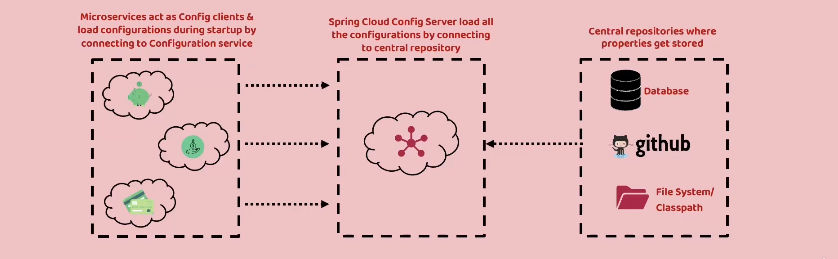
**3** . whenever we try to use environment variables they lake granular access like whoever has access to our production server like all the server admins we see our environment variable may be we are trying to config the database credentials or some senetive properties with the help of environment variables bcoz we don’t want to mentions them inside the application.yml file or inside the profile file directly since that is going to expose our sensitive info to everyone but even we try to follow the environment variable with this approach also our server admin will be seeing our environment variable that why this is consider as a series security limitation inside the microservices we will deploy multiple instances of a single microservice like we have there microservices like accounts ,loans and cards but if we decided to deploye three instances of each microservices than there will be total nine instances will be running inside our production in such scenario providing all these externalize configuration with the CLI argument jvm properties or any other approach is going to be super super challenging and here just talking about three microservices where each one has three instances think like 100 microservice where each has atleast 5 instances then the number of instances are going to run inside our production is going to be 500 hunddred instances we can imagine even if we have a single manual task involve for a microservice is the same has to be repeated in all he microservice that why this approach is not going to work when we are trying to deploy multiple instances of our microservices ……..WE can overcome this challenges using the **spring Cloud Config**

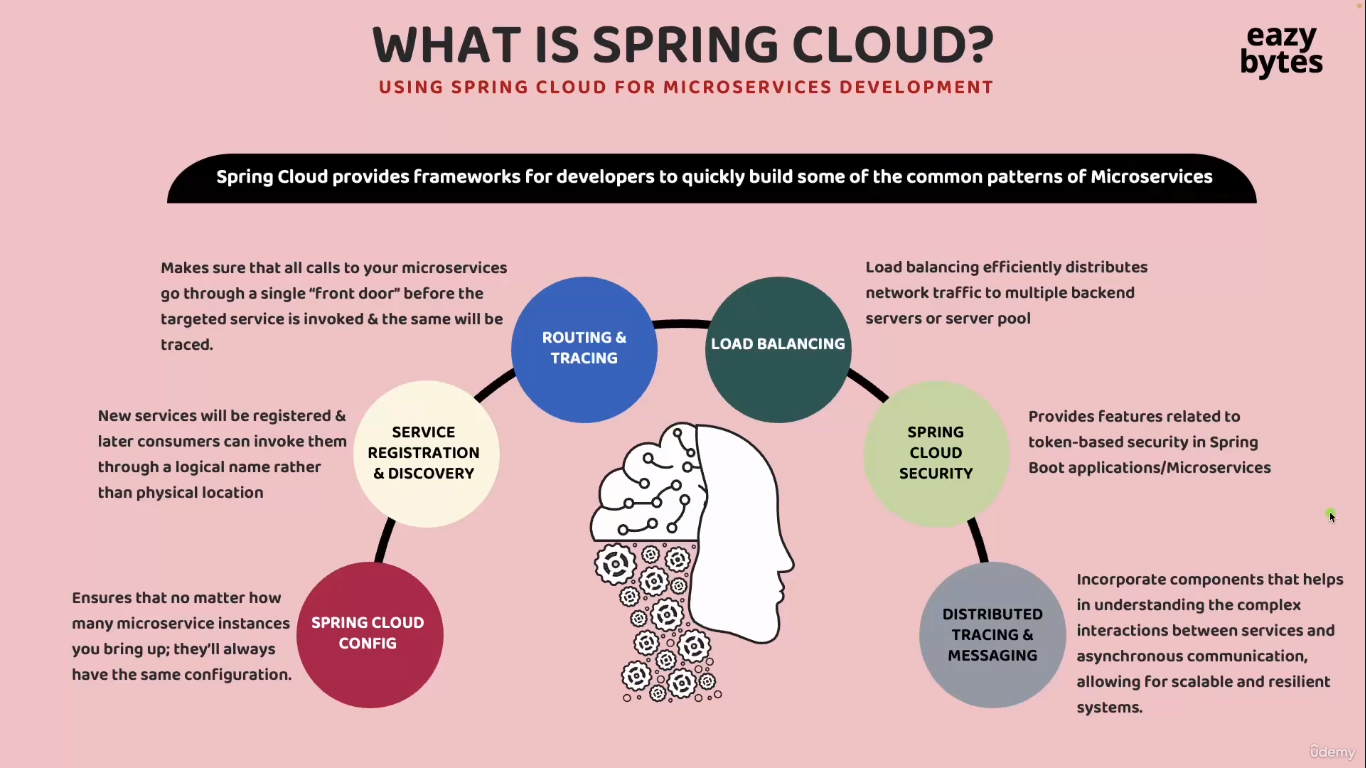
**Introduction to Spring Cloud Config**

To handle the configuaration inside any cloud native sys like microservices or any cloud application that we r trying to build in this approach we are going to have a centeralize configuration server which means we need to build a separate application that is going to act as a configuration server with the help of spring cloud configuration using this centralized configuration server we can overcome all the limitation and drawback that we discuss in the previously and whenever we are using this approach our centralize configuration server is going to provide server and client side support for externalize configuration in a distributed sys like microservices which means all our individual microservice they can register as a client with the spring cloud config serve rand this spring cloud config server can act as a centralized configuration server it will be more clear when we start implementaing this inside our microservices …

This configuration server which we are going to build with the help of spring cloud config it is going to act as a central place to manage all our external proeprties or configuration for all our mircoservices acroos all the environment the centralize config server that we are going to build with the help of spring cloud config revolves around to core elements

1. We are free to store all our configuaraiton or property file inside any location like we can store inside github repo or inside a file sys or inside a database we choose a location where we want to store all our configuration and properties securely once we store all our properties or configuration then this configuration
2. server is going to oversees the configuaration data with in the datastore facilitating its management distributing to multiple applications like microservices





**Building Config Server Using Spring Cloud**

1. Create a new springboot project with the help of spring initializer and add  
    **config server** and **springboot actuator** dependency

We have 2 different verison inside our pom.ml file but we have two different version for both springboot and config server bcoz springboot and spring cloud are two different project inside the spring eco sys they will their own version number so when we try to create a springboot project from starter sprig.io the automatic mapping of the springboot version to the spring cloud version will happen automatically inside this website and it is also going to add a dependency to add this spring cloud dependency along with the version that we have defined inside the top (under config server)

If we dont use spring.io and having already created springboot project and want to add spring cloud related dependency to an existing springboot application In such scenario we can go to the below link   
<https://spring.io/projects/spring-cloud> they mention the version compatibility with the springbooot version

1. And open the main class of config server application add the annotation to enable the config server **(@EnableConfigServer**)
2. Go to application.properties and we will change to yml bcoz we are using the yml format for this project and open this server-config app **application.yml** file and write the cofigurtion like below
3. server:  
    port:8071

**we** config server running inside our microservice n/w but as of now there no place for our config server to read the configuration that’s why we need to move all the required configuration of our microservices to a centralize location and we need to configure that centralize location inside this config server so that our config will start reading from the centralize location and there r multipleo option that we have when we try to store all our configuration inside a centralize location write from classpath to file sys ,github repository database and the sme time we can also store the inside cloud like aws ,s3 there are manyi option but inside this project we are going to learn three approaches

1. **will store all the configuration of the microservices inside the classpath of this config server**
2. **we can store inside a file sys. We can store any where insider our server or insider our local sys and we can read the configuration from the particular folder insider our file sys**
3. **store with the help of github**

**Reading configuration from the class path location of config server**

Create a **config** folder inside the resources folder under the confiserver microsrvice where we will store the configuration of all of the microservices configuration(loans,cards,accounts) and create the three .yml files for all the microservices which is inside our project (.yml,prod,Qa) we have three microservices inside our project so will creating three files for our all three microservices inside the **config** folder

|  |  |  |
| --- | --- | --- |
| loans-prod.yml | loans.yml | loans-qa.yml |
| build:  version: "1.0"  cards:  message: "Welcome to MsBank cards related prod APIs "  contactDetails:  name: "Sandra Harald - Product Owner"  email: "sandra@Msbank.com"  onCallSupport:  - (617) 432-2356  - (936) 564-872 | build:  version: "3.0"  loans:  message: "Welcome to EazyBank loans related local APIs "  contactDetails:  name: "Amaal Grega - Developer"  email: "amaal@eazybank.com"  onCallSupport:  - (452) 456-2176  - (546) 764-8934 | build:  version: "2.0"  loans:  message: "Welcome to MsBank loans related QA APIs "  contactDetails:  name: "Cyrano Marita - QA Lead"  email: "cyrano@msbank.com"  onCallSupport:  - (785) 545-6565  - (853) 546-3467 |
|  |  |  |
| **cards-prod.yml** | **cards.yml** | **cards-qa.yml** |
| build:  version: "1.0"  cards:  message: "Welcome to MsBank cards related prod APIs "  contactDetails:  name: "Sandra Harald - Product Owner"  email: "sandra@Msbank.com"  onCallSupport:  - (617) 432-2356  - (936) 564-8721 | build:  version: "3.0"  cards:  message: "Welcome to EazyBank cards related local APIs "  contactDetails:  name: "Dragos Lech - Developer"  email: "dragos@eazybank.com"  onCallSupport:  - (412) 419-3491  - (915) 382-1932 | build:  version: "2.0"  cards:  message: "Welcome to MS Bank cards related QA APIs "  contactDetails:  name: "Cherryl Pankaj - QA Lead"  email: "cherryl@msbank.com"  onCallSupport:  - (310) 875-4367  - (201) 236-1267 |

|  |  |  |
| --- | --- | --- |
| **accounts\_prod.yml** | **accounts.yml** | **accounts\_qa.yml** |
| build:  version: "1.0"  accounts:  message: "Welcome to MsBank accounts related prod APIs "  contactDetails:  name: "Reine Aishwarya - Product Owner"  email: "aishwarya@msbank.com"  onCallSupport:  - (453) 392-4829  - (236) 203-0384d | build:  version: "3.0"  accounts:  message: "Welcome to EazyBank accounts related local APIs "  contactDetails:  name: "John Doe - Developer"  email: "john@eazybank.com"  onCallSupport:  - (555) 555-1234  - (555) 523-1345 | build:  version: "2.0"  accounts:  message: "Welcome to MsBank accounts related QA APIs "  contactDetails:  name: "Smitha Ray - QA Lead"  email: "smitha@msbank.com"  onCallSupport:  - (666) 265-3765  - (666) 734-8371 |

Now open the application.yml file of springconfig microservice and write the configurations

spring:  
 application:  
 name: "configserver"  
 profiles:  
 active: native  
 *#active: git* cloud:  
 config:  
 server:  
 native:  
 search-locations: "classpath:/config"  
  
server:  
 port:8071

**active:** native(means we are trying to activate native profile of spring cloud config server)

**spring.profile.config.server.native** :- giving the location where our properties is stored right now our property store in the class path (**classpath:/config)**  
**-------------------------------------------------------------------------------------------------------------**now start the serverconfig app and it is running on port **8071**

**Invoke various get api path that r expose by the config server**

[**http://localhost:8071/cards/prod**](http://localhost:8071/cards/prod) **:- this** url will give the prod file and default file data of cards from the config folder

**{**"name": "cards","profiles": **[**"prod"**]**,"label": **null**,"version": **null**,"state": **null**,"propertySources": **[{**"name": "classpath:/config/cards-prod.yml","source": **{**"build.version": "1.0","cards.message": "Welcome to MsBank cards related prod APIs ","cards.contactDetails.name": "Sandra Harald - Product Owner","cards.contactDetails.email": "sandra@Msbank.com","cards.onCallSupport[0]": "(617) 432-2356","cards.onCallSupport[1]": "(936) 564-8721"**}}**,**{**"name": "classpath:/config/cards.yml","source": **{**"build.version": "3.0","cards.message": "Welcome to MSBank cards related local APIs ","cards.contactDetails.name": "Dragos Lech - Developer","cards.contactDetails.email": "dragos@msbank.com","cards.onCallSupport[0]": "(412) 419-3491","cards.onCallSupport[1]": "(915) 382-1932"**}}]}**

**http://localhost:8071/cards/qa :- this** url will give the qa file and default file data of cards from the config folder

**localhost:8071/cards/default:- this** url will give the default file data of cards from the config folder

So all the profiles are loaded into our spring cloud config server like whatever configuration we have store inside the classpath all of them loading during the startup of config server

**Establish the link between config server and our microservices and reading configuration from the class path location of config server**

Go to the accounts microservices and delete the **application\_prod.yml** and **qa**.**yml**  file bcoz we no more needed them bcoz we are right now storing inside the config server

Now adding the config client dependency Go to spring initializer and search the config client and add into the pom file like below instruction

1. Copy the property and paste after java version like below (bcoz we are adding server config dependency in account microservices )
2. <properties>  
    <java.version>17</java.version>  
    <spring-cloud.version>2024.0.0</spring-cloud.version>  
   </properties>
3. And now add the dependency management after dependencies tag and dependency also

<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>

We need to communicate accounts microservice about the end point url details of config server . That’s why we need to go to the application.yml of accounts mircroservice

Here we need to create an property which is **spring.config.import**

spring

config:  
 import: "optional:configserver:http://localhost:8071/"

we are using optional becoz if we are not able to connect(or our config server is not started before the account microservice we can see our client microservice) with the config server for whatever reason it can still continue start the microservice application ….if we have important properties inside config server without them we can not proceed then we can remove the optional from here .

**accounts(application.yml)**

server:  
 port: 8080  
spring:  
 application:  
 name: "accounts"  
 profiles:  
 active: "prod"  
 datasource:  
 url: jdbc:h2:mem:testdb  
 driverClassName: org.h2.Driver  
 username: sa  
 password: ''  
 h2:  
 console:  
 enabled: true  
 jpa:  
 database-platform: org.hibernate.dialect.H2Dialect  
 hibernate:  
 ddl-auto: update  
 show-sql: true  
 config:  
 import: "optional:configserver:http://localhost:8071/"

**spring:  
 application:  
 name: "accounts" :- this is our accounts microservice name and all the properties file related to accounts which is stored inside configserver app should have name accounts.yml or accounts-prod.yml or accounts-qa.yml**

previously we are added all the yml file(prod ,qa, for all the three microservices accounts and loans and cards) inside the resource of config server microservices

**configserver.yml**

spring:  
 application:  
 name: "configserver"  
 profiles:  
 active: native  
*# active: git* cloud:  
 config:  
 server:  
 native:  
 search-locations: "classpath:/config/"  
  
server:  
 port: 8071

**search-locations: "classpath:/config/" :- T**his is defining the path means all the properties files of all three microservices are in the config folder (under resource of configserver micro service)

**name: "configserver" :**using this name we can configure in the other microservice from where we want to access the properties files   
like in accounts we are connecting the config server using this name in previous page(client microservice where we want to connect configserver microservice to fetch the centralize properties files)

configserver:http://localhost:8071/"

now we are ready to read the properties from the config server   
Hit :- [**http://localhost:8080/api/contact-info**](http://localhost:8080/api/contact-info)

**output :-** {

    "message": "Welcome to MSBank accounts related prod APIs ",

    "contactDetails": {

        "name": "Reine Aishwarya - Product Owner",

        "email": "aishwarya@msbank.com"

    },

    "onCallSupport": [

        "(453) 392-4829",

        "(236) 203-0384"

    ]

}

**This is coming from the config server** we are getting the prod proeprties bcoz we are activating the prod profile in the accounts microservice property file

But if we try to activate the different profile from external configuration and see our microservice is able to fetch the relevant profile properties from the configserver microservice   
Right click on microservice app and click on MoreRun->Modify run configuration and write inside argement input box :- **--spring.profiles.active=qa**

**Now we can see our qa profile active and we are able to fetch the qa propeties**

{

    "message": "Welcome to MsBank accounts related QA APIs ",

    "contactDetails": {

        "name": "Smitha Ray - QA Lead",

        "email": "smitha@msbank.com"

    },

    "onCallSupport": [

        "(666) 265-3765",

        "(666) 734-8371"

    ]

}

**Reading configuration from a file system location**

And here you may have a question like why some projects they prefer to use file system approach.

**Using file system approach**.

You are going to maintain all the required properties inside the server location where your microservice is deployed. And this approach has an advantage like your server admins, they can enforce some security restrictions.

Like no one can open that folder and no one can see the content of that folder except your config server application.So that's why few projects they follow this approach as well.

**Now let's try to understand how to store all our configurations inside a file system location.**

So behind the scenes what I have done is I have taken all the files inside my config folder.You can see all these files that I have highlighted. I have copied to a location inside my local system.So this is the location where I have copied.

And here if I try to show you the path where I have stored, you can see I have stored under the **users/ms/documents**, config. So this is the folder location like where I have copied.If I try to navigate outside there is a config folder and this config folder is present inside the documents.And this documents is present inside the ms and this ms is present inside the users,which is the root location inside my system.So now I have the configurations successfully copied into a file system.As a next step I need to open the application.yml inside my config server.As of now you can see we have mentioned the search location as classpath.Instead of this we need to provide a new search location.

Here I'll try to comment this existing one for your reference post that I'm going to copy the same search location property and mention the same here inside the native element.So now here instead of classpath path, I should mention file after the file I need to give the colonand after the colon, I need to make sure I'm giving three forward slashes followed by what is a folder path or the file system path where I have stored my configurations inside my local system or inside my local server.So you can see user, ms, documents, config.

This is the folder location**.:- "file:///C://Users//User//Documents//config"**

And we need to make sure we are mentioning two forward slashes between every folder and after this file colon prefix, we need to make sure we are mentioning three forward slashes.

With this now my config server will point to a new search location and this new search location is following the approach of file system.

Now I can stop all my microservices including the config server and do a build. As a next step,first I can start my config server because like we know before we try to start all the individual microservices we need to start the config server.So the config server started successfully post that

I'm going to start my accounts application followed by loans application and at last cards application.So all my microservices are also getting started successfully.First, I can quickly try to test my config server URLs. So first I'll try to test prod profile of loans microservice. **http://localhost:8071/loans/prod**

I'm able to fetch the properties.You can see the location it is showing as

**file:/C:/Users/User/Documents/config/loans-prod.yml.**

This confirms we are successfully able to load the configurations from a file system location.Very similarly, I can randomly test for cards as well, followed by accounts as well.So accounts microservice profile is also getting loaded properly.Now as a next step, let's try to test the integration between our individual microservice with the config server. I'm not going to test all the APIs. Let's try to test randomly for cards Microservice

by invoking this contact-info, you can see we are getting the prod related properties because by default for all of our microservice we have activated the prod profile.I hope you are clear with the changes that we have done inside this lecture to read the configurations from a file system location.The only change that we have done is to change the value inside the search location from classpath to file along with the folder location(putting the folder in our local sys directory like c driver or e or anywhere in the sys).The profile has to be native only inside your spring cloud config server.

spring:  
 application:  
 name: "configserver"  
 profiles:  
 active: native  
*# active: git* cloud:  
 config:  
 server:  
 native:  
*# search-locations: "classpath:/config/"* search-locations: "file:///C://Users//User//Documents//config"  
  
server:  
 port: 8071

**Reading configuration from a Github Repository**

GitHub repo. Using the same GitHub repo,

my config server can try to load and read all the properties during the startup. And this GitHub approach

is the most recommended approach because when you try to store your properties inside GitHub repo you

will get multiple advantages like you can properly secure your GitHub repo. So that no one can access it.

And at the same time it is also going to support versioning, auditing, in future maybe down the line

after one two years if you want to understand what is the property that we use to have two years back or one year back, you can always see the history inside the GitHub repo, whereas other approaches like file system and classpath it is nearly impossible to track such versioning changes.

So in order to get started with the GitHub approach, first we need to move all our properties into

a GitHub repo. Behind the scenes I have created a GitHub repo.You can see this is the GitHub repo with the name eazybytes-config. Inside this GitHub repo,like you can see, I have uploaded all the properties related to the accounts, cards and loans microservice.And apart from that you may also see other Yaml files like Eureka server, Gateway server.So for now don't worry about them.We are going to use them in the coming sections.So please focus on only the accounts, loans and cards microservice Yaml files.

If you try to open any of them, the same content is present, whatever we have discussed previously.

So now you can see as of now I made these repository as public so that my students can access these

and they can also use inside their microservice while they are practicing.In real world,we can't make our GitHub repo as a public.Instead, we are going to make it as private repo, which means our config server has to authenticate with the GitHub repo whenever it is trying to connect during the startup.

I'll give some directions on how to provide security details,whenever you are trying to interact with a private GitHub repo from your config server.For now first let's try to connect with this public GitHub repo. For the same inside your public GitHub repo,first, please copy the value of https URL.

So this is my https url value.So I'm just copying this post that we'll go to the application.yml of config server.Here inside the config server as of now we have activated the profile native and this native profile

we should activate only when you are using the approaches like classpath, location and file system

location.Whereas if we are using GitHub repo we need to activate the profile with the name git.

So that's why I'm going to mention active post that I'm going to mention the profile name as Git.

So please make sure you are mentioning this active under the profiles only. So I have mentioned this correctly.Like you can see this is the child under profiles. I just commented this so that it will be here for your reference.

Now under the spring.cloud.config.server.native.We have provided the search locations, but right now we are not following the native approach or native profile.That's why.Please comment all these native related configurations.I'm trying to comment the search locations as well. Now inside this server, let's create a new child element with the name Git.You can see the same value we have to provide here.

Under the git we need to create one more child element, which is uri. To this Uri,

we need to pass the GitHub repository URL details.So this is my GitHub repository URL details.

After mentioning these details, we should also mention one more child element under the gate which is default-label. Like you know, inside the GitHub repo we'll be having a label name for our branches.

Inside my GitHub repo you can see the default branch name right now I'm using is main.

That's why I have to mention the same.This default label element is going to helpful if you have multiple branches inside your GitHub repo. To avoid such confusion we need to always mention what is the default label or what is the default branch that your config server has to connect. After this property

I'm going to mention one more property which is timeout. Under this timeout I'm going to give a value as five.So this represent that my config server should wait only for maximum five seconds and after the five

seconds if it is not able to connect to my GitHub repo for whatever reasons, the config server should immediately throw an exception.So this will allow my config server to fail immediately and that will give an exception to the operations team or the developers team to understand if the GitHub repository is down or if the configurations that we have done related to the uri or any other properties is wrong or not, we can validate all such details. After the timeout,the next property that we need to mention is, clone-on-start.

Start with the value

true.With this property I'm telling to my config server.Please try to clone the GitHub repo into local system during the startup itself.If you don't mention these value as true, the cloning of the GitHub repo will happen only when the very first request come to your config server and this may result into some issues like your config server may get started properly, but when the very first request coming towards your config server like in the form of accounts microservice or cards or loans microservice, then it cannot clone and it cannot read the configurations.And in such scenarios it is going to create some issues on the accounts microservice.That's why we need to make sure we are cloning the GitHub repo during the startup of the config server itself. And the last property that I want to provide here is, force-pull.

So to this force-pull. I'm going to mention the value as true. Sometimes you might have changed some local changes inside the local repo that got cloned inside your config server. By mentioning this force-pull as true. We are telling to override all the local changes whenever you are trying to start or restart

your config server. This will make sure your config server always reading the properties from the master location which is GitHub repo.After making these changes we should be good from the code changes perspective.

spring:  
 application:  
 name: "configserver"  
 profiles:  
*# active: native* active: git  
 cloud:  
 config:  
 server:  
*# native:  
# search-locations: "classpath:/config/"  
# search-locations: "file:///C://Users//User//Documents//config"* git:  
 uri: "https://github.com/Shakir-Ali01/msbytes-config.git"  
 default-label: main  
 timeout: 5  
 force-pull: true  
server:  
 port: 8071

Let me save this changes and do a build post that I'm going to stop all my microservices including the config server. After all my services are stopped. As a next step I can first try to start my config server application.So let me start in debug mode and you can also check in the console that your config server will try to use the git profile and that's what happened here.

Now as a next step, I'm going to start my AccountsApplication. Once the AccountApplication is started, I'm going to start my LoansApplication and after the LoansApplication I'm also going to start the CardsApplication. With this we have started all our services.First, let's try to understand if our config server is reading from the GitHub repo or not. Here I'm trying to load this URL, which is [**http://localhost:8071/loans/prod**](http://localhost:8071/loans/prod)

**Output**

**propertySources": [**

**{**

**"name":"https://github.com/Shakir-Ali01/msbytes-config.git/loans-prod.yml",**

**"source": {**

**"build.version": "1.0",**

**"loans.message": "Hey, welcome to EazyBank loans related prod APIs ",**

**"loans.contactDetails.name": "Pelias Sudhir - Product Owner",**

**"loans.contactDetails.email": "pelias@eazybank.com",**

**"loans.onCallSupport[0]": "(723) 656-8709",**

**"loans.onCallSupport[1]": "(156) 342-0956"**

**}},**

Here you can see as soon as I refresh the page I'm able to get the properties from the GitHub repo, the same you can confirm by looking at this GitHub url link that my config server is using to pull the properties. So these are the accounts prod related properties.

As a next step, let's try to test the integration between the individual microservices and config server.

Here I'm going to invoke the cards related contact-info. You can see I'm getting a successful response from the cards microservice, which means my cards microservice during the startup it might have connected to the config server and asked for the properties related to the prod profile.

With this right now our config server is pointing to the GitHub repo and this is the most recommended

approach and this approach has many other advantages which we are going to discuss in the coming lectures.Throughout this course,we are going to use this GitHub approach. Now you may have a question here.Are there any other approaches that we can use because different projects may have different requirements.So let's try to understand the same by looking at the official documentation of the spring cloud config server.Here inside the spring website, you can click on this spring cloud config.

So I'm just clicking on this which will open the spring cloud config related information.

Now in order to check the official documentation details, you can click on this learn and post that

you can click on this reference doc of the latest version.With that, you will be redirected to the official documentation.If you can click on this spring cloud config server and scroll down there is a lot of information about the spring cloud config server. If you ask me to talk about spring cloud config server, I can talk for ten more hours.So this subject is very huge.That's why if you have any questions, any time, I would always recommend you to come to this page and check for the information available about the spring cloud config server. Suppose if you are trying to look for the information on how to use a private GitHub repository, then you can come to this git backend and here there'll be a section explaining about authentication. You can see and this authentication we have the information on how to pass a username and password of your GitHub repo. But if you are not comfortable mentioning the username and password, you can also make this work by following the SSH standards. So please read this official documentation for more details. Now, very similarly, if you are using AWS codecommit just like how we are using GitHub repo in such cases you can refer to this section and similarly Google cloud source

if you are using you can refer to this section and git SSH configurations you can achieve with the help of these properties. Like what is your what is the host key? What is the host key algorithm?

What is your private key. Post that if you can scroll down, we have various approaches explained like

file system backend, which we already discussed. And if you are using some vault, you can refer to this vault backend.

And similarly, if you can scroll down, there is also information about CredHub server, AWS secret manager and very similarly parameter store Jdbc backend in case if you have plans to use database to store all your properties, you can refer to these Jdbc backend. This way this official document has a lot of information. My request is whenever you have some requirements which we have not discussed inside this course, this documentation can be a great resource for you to get started. I want you to be empowered always. Like you don't have to depend on others. You can always refer to the official documentation to get most of the information. Whatever courses that you find inside Udemy or YouTube or any other place, they will only make you the subject simple and they will try to explain the concept.

Once you are clear with the subject and concept, any complex scenario that you have, you can always

try to achieve by referring to the official documentation most of the times. That's how you can try to get promoted yourself from junior developer to senior developer. Always remember your number of years of experience doesn't matter. The knowledge that you carry only is a matter. I hope you are clear with all the information that we have discussed inside this lecture.

**Encryption & Decryption of properties inside config server**

As of now, we have all the configuration properties inside the GitHub repo and right now we are in

a good position because we are following the most recommended and production standard approach. But sometimes we may want to store our properties in an encrypted format.

Currently inside our GitHub repo we stored all the properties in a plain text.

What if you have a scenario where you want to store the property value in an encrypted format so that even if someone has access to your GitHub repo or if someone by accidentally open your GitHub repo, they should not be able to see your sensitive property details.

That's why it is always advisable to encrypt sensitive properties like passwords or any URL details,

folder structures or any other sensitive information.

Inside this lecture, let's try to understand how spring cloud config server is going to help in this

scenario. For the same, first, we need to go to the application.yml of config server. Here we need to create a property with the name encrypt. Post that encrypt,we need to mention a child element with the name key. So to this encrypt.key we need to provide what is the secret that my spring cloud config server canuse to encrypt the properties and to decrypt the properties?

Always remember whenever you are trying to provide a secret key, it has to be super complex so that it is going to be very tough for the hackers to guess it. If you just keep some simple keys like one, two, three, four, five or password or Abcdef. So such keys are very dangerous to use inside any application. We should use a key, which is very complex to guess.That's why here I'm going to mention some complex key.And the key like you can see it is a very complex key and it is really impossible for anyone to guess this key. And this is completely random key.You can create your own key.There are many websites also online which will help you to generate a complex key for your encryption process.So this key can be any value, but please make sure it is complex in nature.

Once we create this property inside our config server, it is going to expose, encrypt and decrypt

related APIs. Using which it is going to encrypt or decrypt all our properties.

So let's try to understand those properties. For the same,

spring:  
 application:  
 name: "configserver"  
 profiles:  
*# active: native* active: git  
 cloud:  
 config:  
 server:  
*# native:  
# search-locations: "classpath:/config/"  
# search-locations: "file:///C://Users//User//Documents//config"* git:  
 uri: "https://github.com/Shakir-Ali01/msbytes-config.git"  
 default-label: main  
 timeout: 5  
 force-pull: true  
encrypt:  
 key: "6SD893HFHJK399JFJFHJF88"  
server:  
 port: 8071

I'm going to save these changes and do a build post that I'm going to stop all my applications and for

now I'm going to start only the config server application. So here, let me start in debug mode.

Once our config server is started, we can go to the postman.

Here inside the postman you can see under the config server folder.I have provided a new request with the name encrypt and this is going to support Http method post and the URL has to be your config server port number which is 8071 and encrypt is the path. To this path

you can pass any kind of plain text value and it is going to give the encrypted value. And this encryption

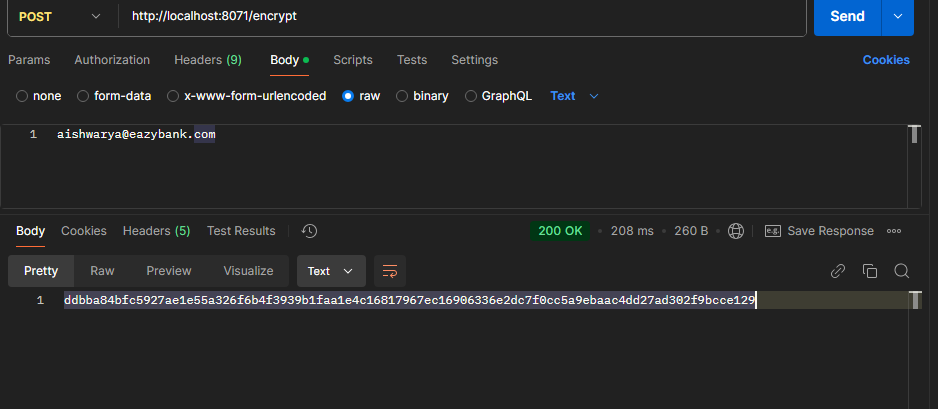
process is going to use the secret key that you have defined inside the application.yml of config server. So here I'm trying to encrypt a property which is the email inside my accounts prod profile.

As of now you can see inside my accounts-prod.yml, under contact details,

the email value is mentioned in a plain text value, so anyone who has access to my GitHub repo they can see the email value.Think like I want to encrypt this value inside my GitHub repo. In such scenarios, first you need to understand what is the encrypted value of your plaintext value.

That's why I'm trying to take this plain text value from here. Now, inside the body of this post request, I need to select this raw option and please make sure you

are selecting this text option because I'm just trying to send a simple text value. So here I have mentioned the email in a plain text. Now if I try to click on the send button,



I will get an encrypted value.Now we have the encrypted value of the plain text value(**aishwarya@eazybank.com**).We can copy this, encrypt value and go to the  
 GitHub repo. Inside my GitHub repo

like you can see as of now we have the email using the plain text value. Since we have the requirement to encrypt this email value, what I can do is, I can click on this, edit

this file, and in the place of plain text value I'm going to mention the encrypted value. Please make sure you also have this double quotes. But here there is a challenge for my spring cloud config server.How it is going to differentiate between a plain text value and a encrypted value. To help spring cloud config server

around this scenario, I need to make sure for the encrypted values I'm mentioning a prefix which is

inside the curly braces, I have to mention cipher.

build:  
 version: "1.0"  
accounts:  
 message: "Hey, welcome to EazyBank accounts related webhook APIs"  
contactDetails:  
 name: "Reine Aishwarya - Product Owner"  
 email: **"{cipher}**ddbba84bfc5927ae1e55a326f6b4f3939b1faa1e4c16817967ec16906336e2dc7f0cc5a9ebaac4dd27ad302f9bcce129"  
onCallSupport:  
 - (453) 392-4829  
- (236) 203-0384

So whenever my spring cloud config server sees this cipher which is a prefix to an value, then it assumes whatever value of the cipher is an encrypted value. So during the when it is trying to send these properties to the actual microservice, it is going to decrypt and send the values in a plain text value. This way, even if someone is trying to see the properties inside my GitHub repo, I'm fine because these are encrypted values.

They cannot know what is the encrypted value until unless they know the secret key that I have mentioned inside the config server. As of now, the secret key we have simply configured inside the application.yml of config server. But in real production applications the secret key can be configured with the help of environment variables or CLI arguments or any other approach.

So now let me commit these changes and commit directly into the master branch. As a next step,

let's try to restart our config server so that it reads these latest values.

So here I will stop my config server first post that I'm going to start my config server in a debug

mode.Once my config server is started, we can validate if the config server is able to decrypt the value or not by invoking the application slash prod API inside the config server.

So here I'm trying to invoke this API which is account slash prod available inside the config server.

You can see I'm getting a response and here the email is decrypted

**http://localhost:8071/accounts/prod**

**output** :-

"**build.version": "1.0",  
"accounts.message": "Hey, welcome to EazyBank accounts related webhook APIs",  
"accounts.contactDetails.name":"ReineAishwary-Product-Owner",  
"accounts.onCallSupport[0]":"(453)392-4829",  
"accounts.onCallSupport[1]": "(236) 203-0384",**

**"accounts.contactDetails.email": "aishwarya@eazybank.com"**

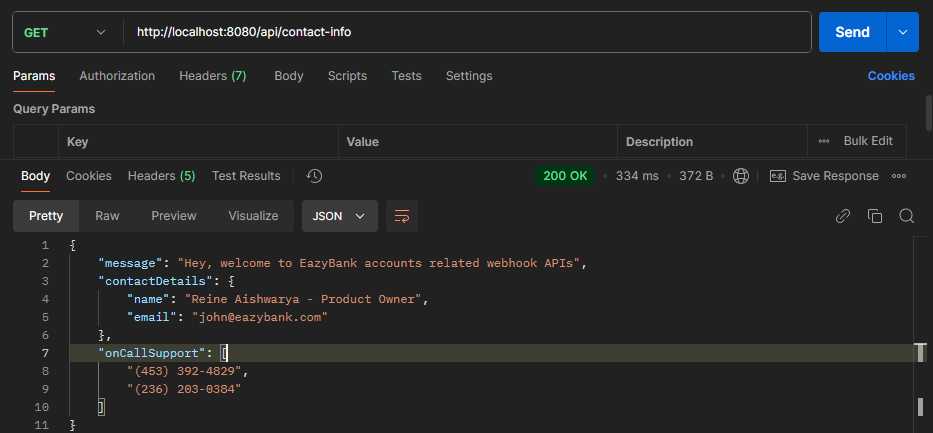
.

So this is the email which we initially encrypted.

By the time config server is returning to the clients, it is going to decrypt with the secret key that

we have mentioned. Now as a next step, let's try to start our accounts microservice and validate if my accounts microservice is receiving this email value in a plain text or in an encrypted value.

So I'm trying to start my accounts microservice in a debug mode and I'll go to the postman. Inside my postman you can see this is the API that we need to invoke to get the contact-info details from the accounts microservice. As soon as I click on the send button, you can see I'm getting the prod profile related properties



because by default the prod profile is activated. And the important check that we need to do here is,

this email is coming in a plain text value instead of encrypted value.

This way we can store any number of sensitive properties inside the GitHub repo using encrypted format.

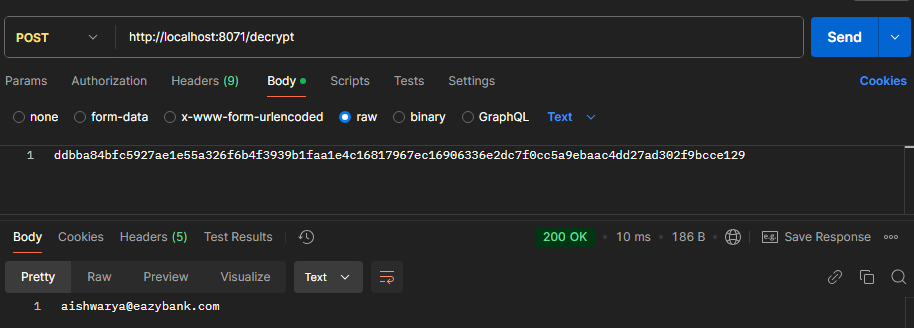
Like I said before, just like encrypted API path, the config server is also going to expose decrypt

API as well. Let's try to test that decrypt by taking this value.

So here I'm going to create a new request and the request will support post and the url is,

**localhost:8071/decrypt**.

Here for this API, we need to go to the body and make sure you are clicking on this raw and the format is text and mention the encrypted value. If I try to click on the send button



you can see I'm getting a plain text value. Here I don't have to mention the cipher before my encrypted value because since we are already invoking the decrypt API, my config server knows that my developer or end user or the client application is going to pass the encrypted value only.

Here again, you may have a question like if someone can easily decrypt my encrypted properties then what is the use of encrypting it? Like you said, inside production applications, your config server will work in a very different manner. Here we are able to invoke any API against config server very easily, but inside production applications, your platform team, they are going to deploy your config server behind the firewalls of your organization. That means no one can invoke the APIs of config server directly .Only the applications which are deployed within the firewall of the organization

they can only communicate with your config server.On top of that, if you want to secure your config server, you can secure it with the help of spring security framework, just like any other spring boot application .I hope you are clear like how to encrypt and decrypt the properties.

**Refresh configurations at runtime using refresh actuator path**

 we have three different microservices and there is a

config server and we are able to read the properties from the config server during the startup of the microservices applications.

So everything is working perfectly and you may think this is the end of the spring cloud config server and we may not face any other challenges in terms of configuration management. But inside this lecture I want to introduce a new problem that we may face inside the microservices environment in terms of configuration management. Think like you have your config server setup and all your microservices started by connecting to the config server they loaded the properties perfectly. All of a sudden you want to change a particular property inside your config server and the same you want to be reflected runtime without restarting your microservices(bcoz without restarting microservices changes will not reflected we need to restart our m icroservices to get changes properties from the config server ).

**Here you may have a question, what is the problem that I have if I restart my microservices**.

Inside microservice It's not one microservice there are hundreds of microservices and there will be multiple instance for each microservice. So restarting your instances of microservices is again a manual task that someone has to take care. Whenever you bring some manual task inside microservice, then it is going to make your microservices setup very complex. That's why we should look for an option on refreshing the properties without restarting the microservices instances.

**For example**, think like you have a feature flag which you have configured inside the config server. So based upon a feature flag like a boolean flag, you want to control the behavior of your microservice business logic. When the flag is disabled, you want to execute some other piece of code. These flags you want to change anytime inside the config server and you want the same to be reflected immediately inside your individual microservices without restart.So this is the most common scenario that projects will try to achieve inside their microservices network. That's why inside this lecture, let's try to focus on

**how to refresh the configurations or properties inside the microservices without restarting the instances.**

**Step 1**. :- first we need to make sure all our individual microservices, they have the spring boot actuator dependency defined inside the pom.xml

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

**Step 2:-** As a next step, we need to go to our Dto classes where we are trying to hold all our property details. So here we have inside accounts microservice there is a record class with the name AccountsContactInfo which holds all the properties that my microservice is going to read during the startup from the config server. With this setup we have a problem whenever we are using a record class,

that means once the object of this AccountContactInfoDto is created during the startup. We cannot change the property values at runtime by invoking the setter method. Whenever you are using record class, all your fields are going to be final.  
changing the record to class we can see below

|  |  |
| --- | --- |
| **Class** *@ConfigurationProperties*(prefix="accounts") *@Getter @Setter public class* AccountContactInfoDto {  *private* String message;  *private* Map<String,String> contactDetails;  *private* List<String> onCallSupport; } | **Record**  *@ConfigurationProperties*(prefix="accounts") *public record* AccountContactInfoDto(String message, Map<String,String> contactDetails, List<String> onCallSupport) { } |

So we have made these dto related changes in all the microservices.  
So this will allow our microservices to change the property values at runtime.

After making these changes, I need to go to my application.yml of Accounts Microservice. Inside this application.yml, we need to enable the actuator API paths by default actuator is not going to expose all the management related API paths. That's why we need to specifically enable them by introducing a property here. So the property that I want to mention here is, management since we want to enable the management related

APIs

management:  
 endpoints:  
 web:  
 exposure:  
 include: "\*"

our accoutns application is running on the por no 8080 so   
[**http://localhost:8080/actuator**](http://localhost:8080/actuator)

after hitting this url we can see all the endpoint related to actuator is showing   
becoz we r using \* if we want to start only specific endpoint then specify the name like refresh  **for** refreshing the microservices we have need to use this url which is provide by the actuator [**http://localhost:8080/actuator/refresh**](http://localhost:8080/actuator/refresh)

**this is post endpoint so we will hit inside the postman**

**Step 3 :-** Change the values on the config server and try to get the updated value without restarting our microservices change the message value at github accounts-prod.yml file (prod to production )  
**message: "Hey, welcome to EazyBank accounts related to production APIs"**

Now want to access in account microservices but getting previuos value prod to get the updated value we have to hit this url [**http://localhost:8080/actuator/refresh**](http://localhost:8080/actuator/refresh)

**After refreshing we are getting updated values   
Output**

"message": "Hey, welcome to EazyBank accounts related to production APIs",

**So using this approach we can get updated value without restarting the microservices**

So this is super, super perfect.

But there is a serious drawback that we have inside this approach. The drawback is think like you have 100 microservices and each of them has five different instances,which means there will be total 500 microservices instances running inside your production.And for some reason you're trying to change the property in all the microservices.Then you need to invoke the refresh endpoint against all the 500 instances running inside your production.And doing this manually is going to be super, super cumbersome process.Some operations team are some platform team,they will try to automate this process by writing some scripts inside the CI/CD pipelines or they will try to write a Jenkins jobs or CI/CD jobs, which will invoke all the microservices instances, refresh and points. But still, it may not be a convenient solution for many projects. That's why let's explore this further and try to identify is there any better option that we have to refresh the properties dynamically without invoking this refresh endpoint for each and every microservice

**Refresh configuration at runtime using Spring Cloud Bus**

Right now we set up our microservices with config server, but the problem that we are facing is,we have to invoke the refresh API for each microservice instance. Whenever we are trying to refresh the configurations at runtime without restart. So to overcome this challenge we need to use a new project inside the spring cloud, which is spring cloud bus. So whenever you are using the spring cloud bus behind the scenes this spring cloud bus is going to interlink all your microservices instances with a lightweight message broker like Rabbitmq or Kafka. With this, the advantage is you need to invoke a bus refresh api path available against your actuator only one time for one of the instance. If there are 500 instances running inside your production, you don't have to invoke the actuator refresh api for all your 500 instances. Instead, you can simply invoke bus refresh API for any of the instance inside this total 500 instances. With that, the spring cloud bus will take care of communicating the changes happened on the spring cloud config server to all other nodes are the instances connected to the same message broker like Rabbitmq?So in order to get started with this approach, first we need to have a rabbitmq running inside our local system

**Step 1** :- So in order to get started with this approach, first we need to have a rabbitmq running inside our local system. So goto rabbit mq website download and install it https://www.rabbitmq.com/docs/download

.The very first and the easiest option that we can follow is, if you run this Docker run command

**docker run -it --rm --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:4.0-management**

this Rabbitmq message broker is going to run inside your local as a Docker container.We already have Docker installed inside your system, so this is going to be a easiest option to install and set up Rabbitmq. You can see the power of Docker here.Without Docker, you need to download it, you need to install it, you need to start the Rabbitmq. So with the help of Docker, we are going to and avoid all those installation steps.So let me copy this command and I'll go to the terminal. Inside my terminal,I'm going to run this command.

Whenever you are trying to set up Rabbitmq behind the scenes, it is going to install two components.One is a management component which which is responsible to manage your rabbitmq and provide the UI to the Rabbitmq.

And the second one is a core component which is going to handle all the message-q related functionality. So we don't have to type these, we can simply get this command from the Rabbitmq website. Now I'm going to execute this command with that inside the same terminal. The Rabbitmq will start as of now

It started

**Step 2** :- let's go to our workspace inside the pom.xml of all the microservices, including the config server. We need to add a dependency related to the spring cloud bus and rabbitmq. So let me go to the pom.xml of accounts microservice first. So inside this pom.xml here I'm going to add a new dependency just after my spring cloud starter config, so you can see I'm pasting a new dependency.

This is the dependency. The dependency name is spring-cloud-starter-bus-amqp. With this we are trying to set up both spring cloud starter bus and message queue with Rabbitmq integration.Now let me try to add the same dependency inside all other pom.xml.

*<!-- https://mvnrepository.com/artifact/org.springframework.cloud/spring-cloud-starter-bus-amqp -->*<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-bus-amqp</artifactId>  
 <version>4.2.0</version>  
</dependency>

After adding this dependency inside all the microservices instances along with the config server

**Step 3** :- we need to make sure we are enabling the actuator API path which is bus refresh inside all our microservices instances. As of now, if you go and open the application.yml of accounts microservice or any other microservices, and add them below lines (properties )

management:  
 endpoints:  
 web:  
 exposure:  
 include: "\*"

if we want to expose only bus refresh then **include:"busrefresh"**

we have mentioned a property to expose all type of management endpoints.So that's why we don't have to make any more changes here because since we have mentioned this asterisk, it is going to enable and expose refresh path, bus refresh path and any other parts available inside the actuator. The same property we have mentioned inside the cards and loans. So we should be good with this property perspective.

**Step 4 :-**  as a next step, we need to provide the connection details of rabbitmq inside the individual microservices instances. We have all the setup needed and we also started the rabbitmq.

rabbitmq:  
 host: "localhost"  
 port: 5672  
 username: "guest"  
 password: "guest"

But we need to establish a link between the microservices and the message queue by providing the connection details of the rabbitmq. For the same, inside the application.yml under this spring element, you can create a new child element with the name Rabbitmq. Under this rabbitmq mention the host value.

The value of host inside our local will be localhost followed by what is a port number. The port number where we have started the rabbitmq is 5672.

After the port we should pass what is a username by default the username is guessed after username we should also mention what is the password. The password also by default is guessed. So with these properties we should be good. The same set of properties we need to mention inside the cards and loans microservice as well. So let me go to the cards application.yml and here I'm going to copy the same set of values which is rabbitmq, host, port and username and password. As a next step, I can do the same inside the loans microservice as well, so I'm opening the application.yml just after this config.

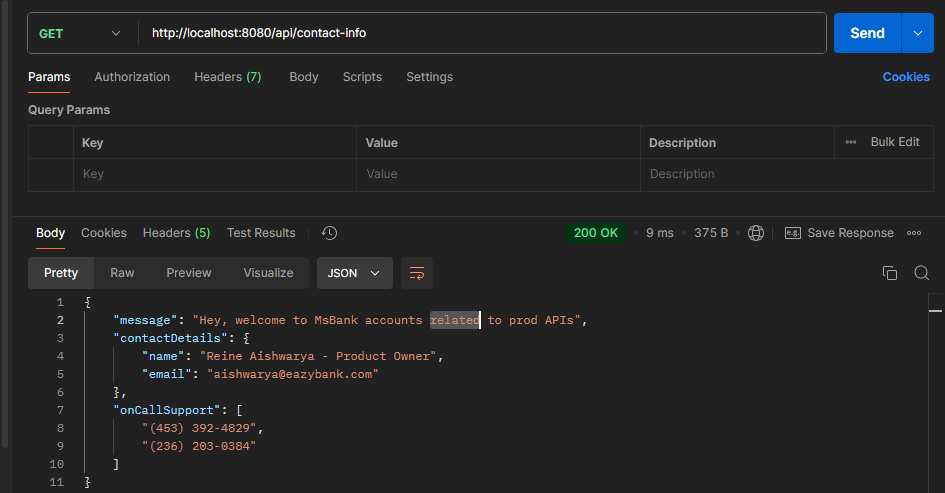
I'm mentioning the rabbitmq connection details. Even if you don't mention these properties still the connection will be established because these are the default values where my rabbitmq is going

to be started. My spring boot is smart enough to consider the default values to connect with the rabbitmq, but I just mentioned here, for your information, in case inside your project, if you are using different port number and different username and password, then we need to mention these connection details inside the application.yml

**Test the functionality of cloud bus is working or not after all the configuration**

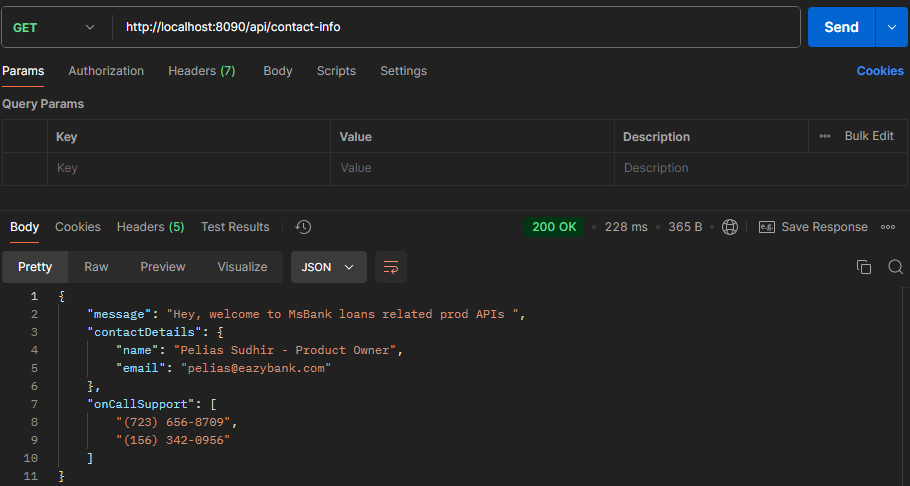
**Changing** the prorerties of all the microservices prod.yml file and want to fetch it is reflecting or not on one bus refresh   
I have chage production to prod in message and I hit the bus refresh for the account mircoservices ….. http://localhost:8080/actuator/busrefresh

After hitting this we got the updated propertis value for account microservices



**Now we are trying to get updated values for loan and card without refreshing**

Boom we can see we are able to get the updated value of loans mircoservices

****

**Refresh config at runtime using Spring Cloud Bus & Spring Cloud Config monitor**

As of now, inside our microservices network, we are able to refresh the properties runtime without restarting our microservice instances by invoking the bus refresh or refresh APIs available inside the actuator.

They both have manual approach involved. Someone has to invoke either bus refresh on any of the one instance or refresh API in all the microservices instances.

But here we are looking for an automated approach which will automate the refreshing of the properties without invoking any API path manually. But the same.There is an option with the help of GitHub webhooks, so this approach is built on top of the spring cloud bus approach that we have discussed in the previous lecture. So all the changes that we have discussed in the previous lecture has to be there to use this approach as well. To get started with the new approach, first we need to add a new dependency inside the palm dot XML of the config server, and this new dependency is Spring Cloud config monitor. We need to add this dependency only inside the config server, but not in the other microservices. Whenever we add these dependency inside the config server, it is going to expose a new API path with the name slash monitor. So this is not a API path under the actuator. This is a rest API path which exposed by the spring cloud config server only, but not the actuator. Using this monitor API path available inside the Spring Cloud Config server, we can create a webhook

inside the GitHub repo saying that whenever a change happened inside my GitHub repo, like a new property is added, please invoke the monitor API path.

As soon as this monitor API path receives a webhook request from the GitHub repo behind the scenes, it is going to invoke the refresh event with the help of Spring Cloud Bus and RabbitMQ.

I hope you are clear with the high level introduction.now will start the step to achieve this automatic refresh

**Step 1:-** inside config server add below dependency

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-config-monitor</artifactId>  
 <version>4.1.1</version>  
</dependency>

**Step 2** :- add properties inside the config server application.yml file

we need to make sure we are mentioning the property which is

management:  
 endpoints:  
 web:  
 exposure:  
 include: "\*"

Because like I said, whenever the GitHub repo send a webhook request to the monitor API behind the scenes, my config server is going to use the Spring Cloud bus. So that's why since behind the scenes it is going to invoke automatically the bus refresh API path, we need to make sure we are enabling all these management endpoints inside the config server as well

**Step 3 :-** we can also mention that RabbitMQ configuration details just to be consistent with all the microservices. Previously I said it is completely optional because as of now we are starting our RabbitMQ with the default values.

So let me mention the same connection details inside the config server as well.

rabbitmq:  
 host: "localhost"  
 port: 5672  
 username: "guest"  
 password: "guest"

**Step 4** :- we need to go to the GitHub repo and create a webhook ->click on Setting option -> click on webhooks options -> Add Webhooks (**The purpose of the webhook is whenever there is a change happen inside your GitHub repo, your GitHub is going to publish an event to the URL that you have configured**) -> under the payload url we need to give the url where web hook request has to be sent by by my GitHub repo so we are giving

[**http://localhost:8071/monitor**](http://localhost:8071/monitor) **(But do you think this is going to work? Definitely.This is not going to work because we are trying to give a localhost URL. If I give a local host URL, how GitHub server will know where this URL is deployed. I'm not giving any public IP. I'm not giving any domain name. It is clueless and it will eventually fail in a production scenario. Your operations team. They will configure a public IP or domain name in the place of localhost)**

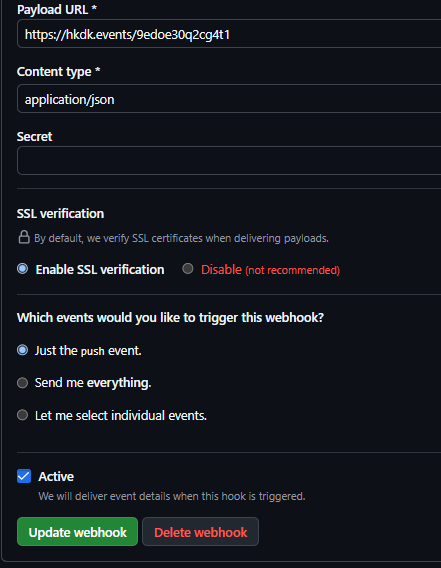
But now how to achieve this in localserver, using hookdeck setup in our sys   
**setup hookdeck step1** :- **npm install hookdeck-cli –g** run this cmd in your cmd prompt make sure your sys having npm packages(setup)

**setup hookdeck step 2:- hookdeck login** now run this cmd(make sure u have account on hookdek website hookdeck.com)

**setup hookdeck step 3**:- **hookdeck listen 8071 --path /monitor** now run this cmd . our config server is running on 8071 and monitor will refresh the cloud bus thing so we are using monitor (monitor dependency exposing the url to refresh is localhost:8071/monitor which is hitting by hookdeck when we will change anything in our repository)after that we will see one prompt What should be your new source label? monitor   
Take any thing u want we give monitor. Now we can see this provide the online url on which our localhost:8071/monitor url will hit/call which is

**Sources**

**🔌 monitor URL: https://hkdk.events/kj7yfbdjhoazj9**

**Step 5 :-** now set this url in the webhook url of github we can see in the image so all the configuration done now is time to check it is working or not now change in your properties file which is on repo so v can see without refreshing anything v r getting updated values Bommm….  


**We can see in the below images how many changes occurs in this image I have edit in two files**

after click on that we can see payload and other info related to this   


**Updating Docker Compose file to adapt Config Server Changes –Part 1**

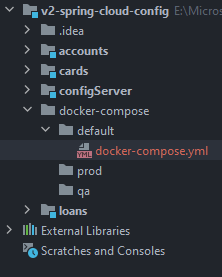
As of now, with the help of config server and other microservices instances, we are able to test all the configuration management related changes inside the local system. But as a next step we need to containerize all our microservices applications, including the config server and post that we should try to start all our microservices along with the config server with the help of Docker compose. Because with the help of Docker containers only, we are going to deploy into the production environments like into Kubernetes cluster. So in order to get started with the continuation of my application, I can straight away generate the Docker images of my microservices and the config server.

But before that I just wanted to write Docker compose files. Once we are clear with the setup of the Docker compose files, then we can try to containerize our applications because we may need to do few more changes inside the config server to make it work inside the docker compose. So that's why first, let's try to focus on preparing the Docker compose files. I can prepare a single Docker compose file, but as of now we have three different environments or three

different profiles like default production and QA to support these three different profiles or three different environments. I want to create different Docker compose files for different environments, so that they will give flexibility to make any changes specific to a particular environment.

So here I'm going to right click on our application and I'm going to create a new directory. The directory name is going to be docker-compose.

Inside this Docker compose directory, I'm going to create three more new directories. The very first one is default and post that the next directory is QA followed by a new directory with the name prod. This way I have three different folders inside my docker compose folder and inside each of them I can create Docker compose file specific to those profiles or environments.



**Now we are going to mention all the details of all the microservices inside the docker-composer.yml like what will be the image name, container name and on which port which microservice will start and all the things related to microservice configuration**

**Docker-compose.yml**

services:  
 accounts:  
 image: "boss215/accounts:s6"  
 container\_name: accounts-ms  
 ports:  
 - "8080:8080"  
 deploy:  
 resources:  
 limits:  
 memory: 700m  
 networks:  
 - msbank  
  
 configServer:  
 image: "boss215/configserver:s6"  
 container\_name: configserver-ms  
 ports:  
 - "8071:8071"  
 deploy:  
 resources:  
 limits:  
 memory: 700m  
 networks:  
 - msbank  
 loans:  
 image: "boss215/loans:s6"  
 container\_name: loans-ms  
 ports:  
 - "8090:8090"  
 deploy:  
 resources:  
 limits:  
 memory: 700m  
 networks:  
 - msbank  
  
 cards:  
 image: "boss215/cards:s6"  
 container\_name: cards-ms  
 ports:  
 - "9090:9090"  
 deploy:  
 resources:  
 limits:  
 memory: 700m  
 networks:  
 - msbank  
  
networks:  
 msbank:  
 driver: "bridge"

now we have three different microservices along with the config server service inside our Docker compose file. Now, as a next step, we need to let our individual microservices and how to connect with these config server inside a Docker environment. To establish the link between config server and individual microservices.we will use the environment properties like below inside all the three microservice

environment:  
 SPPRING\_APPLICATION\_NAME: "cards"  
 SPRIGN\_CONFIG\_IMPORT: "configserver:http://configserver:8071/"  
 SPRING\_PROFILE\_ACTIVE: default

**first one** :- this environment will be write in every microservice which will connect the configserver inside the docker image.this SPPRING\_APPLICATION\_NAME will be same as write in card application.yml in our case this is cards in application.yml of card microservices.

**Though it is the same name which you have defined inside the application.yml.**

**So we're not trying to override anything with a new value.We need to mention this to overcome that bug as a workaround for the defect inside the spring cloud config server. Maybe in future when you are trying, please try without mentioning this and if it is working then that means the bug is fixed in the future versions**  
**second one** :- is connecting the config server with the cards microsevices inside the docker   
[**http://configserver:8071**:-](http://configserver:8071:-) This is config server url inside the docker in which our config server microservice will run  
**configserver** :- **This is prefix to connect any microservice**

**Third one** :- is which profile we will select like we have default , qa , prod so we r connecting the default file

here you may have a question like why can't my accounts microservice or any other microservice,

they can directly use what we have mentioned here.

They cannot use this property because we have mentioned this localhost here.

My accounts docker container tries to use the localhost to communicate with the config server. It is not going to work because my accounts container will start in its own isolated network. When I say localhost, it will try to connect with the config server within its own network, which will never be successful.

That's why we need to externalize this property inside the Docker compose file and we need to override the value present inside this application.yml(**in local sys we are connecting the config server from our microservice application.yml** ) with the help of environment variables. For the same, we can go to the Docker compose file first under the cards microservice, I'm going to mention a new child element. The element name will be environment. Which write in previous page

Now we have the Docker compose file almost ready, but there is a challenge here. As of now, we have a condition inside our microservices which is before the individual microservice try to start. We need to make sure that config server is completely started and ready to accept the request. But with this setup, if I try to run the Docker compose up command, my docker is going to create the

containers in the same order like config server, accounts, loans and cards.

But the problem with this setup is, my docker will not wait for this config server to start completely. It will just initiate the process of the config server and it will immediately jump on to accounts and loans and cards.

And that may create issues because if accounts microservice try to start before the config server is readily available, then it will be an issue.

That's why we need to make certain changes inside this Docker compose file to communicate to the docker and how to identify whether my config server is completely started or not. Once we provided that information, we should also mention that dependency information of config server under all our accounts, loans and cards micro service. That way my docker has a complete information and during the Docker compose command first it will try to start the config server and it will wait for the config server to get completely started and it has good health to accept the request.Once the config server is started then only it is going to start accounts, loans and cards microservice. So to make these changes first we need to understand two important concepts. The very first one is **liveness** and the second one is **readiness**.

**Final docker compose-file**

services:  
 accounts:  
 image: "boss215/accounts:s6"  
 container\_name: accounts-ms  
 ports:  
 - "8080:8080"  
 deploy:  
 resources:  
 limits:  
 memory: 700m  
 networks:  
 - msbank  
 environment:  
 SPPRING\_APPLICATION\_NAME: "accounts"  
 SPRIGN\_CONFIG\_IMPORT: "configserver:http://configserver:8071/"  
 SPRING\_PROFILE\_ACTIVE: default  
  
 configserver:  
 image: "boss215/configserver:s6"  
 container\_name: configserver-ms  
 ports:  
 - "8071:8071"  
 deploy:  
 resources:  
 limits:  
 memory: 700m  
 networks:  
 - msbank  
 loans:  
 image: "boss215/loans:s6"  
 container\_name: loans-ms  
 ports:  
 - "8090:8090"  
 deploy:  
 resources:  
 limits:  
 memory: 700m  
 networks:  
 - msbank  
 environment:  
 SPPRING\_APPLICATION\_NAME: "loans"  
 SPRIGN\_CONFIG\_IMPORT: "configserver:http://configserver:8071/"  
 SPRING\_PROFILE\_ACTIVE: default  
  
 cards:  
 image: "boss215/cards:s6"  
 container\_name: cards-ms  
 ports:  
 - "9090:9090"  
 deploy:  
 resources:  
 limits:  
 memory: 700m  
 networks:  
 - msbank  
 environment:  
 SPPRING\_APPLICATION\_NAME: "cards"  
 SPRIGN\_CONFIG\_IMPORT: "configserver:http://configserver:8071/"  
 SPRING\_PROFILE\_ACTIVE: default  
  
networks:  
 msbank:  
 driver: "bridge"

**Liveness & readiness probes**

**Liveness** is a concept using which we can send a signals from the container or the application indicating whether my container is running properly or it has some health issues. If the output is that the container is alive and it is working properly, then there is no action required because the current state or current health is already good. Whereas if the container is dead, then an attempt should be made by the products like Kubernetes or any other products to heal the application by restarting or by creating a new container. So it is the responsibility of the products like Kubernetes or any other platform to regularly invoke this liveness probe to get the health of my running container.

In simple words, this liveness answer a true or false question. The question is, is this container alive? If the output is true, that means there is no action is required. Whereas if the output is false, then a corrective action needs to be taken by restarting or by creating a new container. If you try to correlate this liveness with a real life scenario inside the boxing, you might have seen a boxer will sit and wait for the fighting to get started. This confirms that this boxer is alive and he is about to start the game. But this doesn't mean he is ready to face the fighting with other boxer. He just trying to warm up himself and trying to take instructions from the coach or he is trying to take the water. So whenever he is in this state, always think like it is indicating liveness

a readiness probe used to know whether the container or app that we are trying to probe is ready to start receiving the network traffic from its clients. Sometimes, especially during the startup, your container might be alive. The output from the liveness probe can be positive, but it may not be ready to accept any new traffic. It might be doing some background work or it might be warming up to accept the request behind the scenes, it might be doing some database initialization. So to get ready and accept the traffic, your container will take more time, especially during the startup time.

So that's why to avoid any scenarios where my Kubernetes or any other platform trying to send the request before it is completely ready, we can make this readiness probe to send an output saying that I am not yet ready. Please give some more time.This way we can avoid scenarios where the liveness is given positive response, but the application or the container is not completely started. And this is a very common scenario, especially during the startup of the container.So platforms like Kubernetes, they will make sure both liveness and readiness, they are giving a positive response in order to send the request that it is being received from the client applications.In simple words, this readiness also answers a true or false question, which is, Is this container ready to receive the network traffic?

So if you try to compare the readiness with the same boxing scenario, whenever the boxer is ready for the fighting, he will come from a sitting position and stand in the middle of the court. So that indicates he is alive. And at the same time he is also ready to continue with the fight.

So that's the difference between liveness and readiness. This liveness and readiness you may see in many places inside Docker, Kubernetes, cloud environments. over all these are the general concepts.

Using the same concepts now, we need to make sure our config server started completely and it is accepting the traffic post that only we need to try to start the containers like accounts, loans and cards. So to help in this scenario, Spring Boot has some actuator endpoints expose and these endpoints are available under the health endpoints. So whenever you are trying to invoke the endpoint URL, which is /actuator/health, it will give you the both of these two indicators in the output. But if you are looking only a specific output, then you can invoke the respective endpoint URLs like actuator/health/liveness and similarly readiness. So these liveness and readiness internally give the output based upon the health indicators from the liveness state health indicator available inside the spring boot framework and very similarly readiness state health indicator.

So using these indicators behind the scenes, my spring boot application can expose the health information by using the actuator endpoint URLs

“**actuators/healh/liveness**” & “**actuators/healt/readiness**”

So now as a next step, let's try to enable these endpoints inside our config server. For the same, first, I need to make sure inside the pom.xml of the config server I have the actuator dependency, so I have added this.

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

As a next step we need to open the application.yml file. So I'm trying to open the application.yml of config server and here we need to add few properties to enabled and expose the health related information. This new configurations we need to mention just under the same position where we have mentioned the endpoints, which means whatever new properties that we are trying to mention has to be defined under the management. So we can see here

So you can see I'm trying to create new properties here.The properties that I have created is first using the health element under the management.

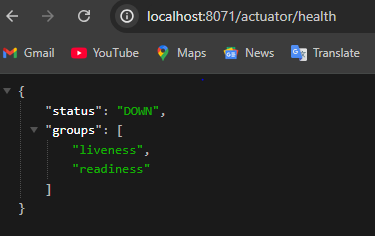
I'm trying to invoke readiness-state and under this I'm trying to invoke one more element which

is enabled and to this enable I'm trying to pass the value true. And similarly I'm also trying to mention the liveness-state and enabled as true with this I'm telling to my actuator, please enable the health related information which will give me the details about the readiness state and the liveness state. And since we want to read these health information using endpoint, we need to make sure we are mentioning a child element under the management. And this child element is endpoint. You can see it is at the same position where we have endpoints.So this is the plural form, this is a singular form.So under this endpoint we need to mention one more child, which is health and under health

there will be props under the props, they will be enabled and the value is true.

.So once we define these properties inside the config server, we can try to test this once inside local.

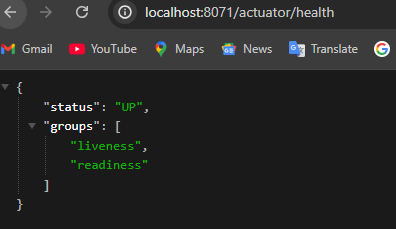
Whether the health information of my config server is being communicated correctly or not with the helpof actuator URLs.



So this will give the overall health details. Like you can see we are getting the status as down. The reason why I'm getting the status as down is, if you try to understand the logs, I'm getting an exception because there is no rabbitmq server or container running inside my local system. The Rabbitmq container that we started previously, I stopped it behind the scenes, so I need to make sure I'm starting this rabbitmq again. So here inside my terminal, I'm running the Rabbitmq related run command

**docker run -it --rm --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:4.0-management**

So once we run this command, the Rabbitmq will start successfully post that my config server also will start successfully. Because now it will be able to connect with the rabbitmq server that we have started. Now my config server might have started successfully. Let's confirm the same. To confirm the same, I'm going to refresh this page which is actuator/health.



**Updating Docker Compose file adapt : Config Server Changes –Part 2**

Now we have our config server exposing the liveness and readiness information with the help of actuator.

As a next step, we need to make some changes inside this docker compose file to communicate with Docker OR docker

compose on how to evaluate whether my config server is in healthy status. For the same, we need to create one more child element under the config server you can create after any element For now I'm trying to create after ports. Which is write below

healthcheck:  
 test: "curl --fail --silent localhost:8071/actuator/health/readiness | grep UP || exit 1"  
 interval: 10s  
 timeout: 5s  
 retries: 10  
 start\_period: 10s

**I'm trying to invoke the API path which is localhost:8071/ actuator/health/readiness.**

So whenever my docker compose, try to start my service config server, it will try to check the health check if it is started successfully or not, by invoking this actuator URL. Once we invoke this actuator URL, we can get the output as up or down since we want to consider only the status up as success. Otherwise we want to consider it as a failure. You can see that's why using this grep command, I'm trying to search inside the response with the value up. If it is there, then it indicates my service is started and its health is perfect. Otherwise I'm telling to simply the exit this health check command, which means my docker compose will

consider that as my config server health is not proper.

Now after this test command, we need to provide more configurations for this health check. So let me try to mention the same.If you leave with this test command alone, my docker compose will only try to run this command only

once, but maybe in the scenarios where my config server might have taken good amount of time, maybe 30s or 20s to get start the complete config server application in such scenarios will always get the output as failure.

That's why we need to provide some metrics like you can see here just under the same test, we need to mention the other elements like interval, timeout retries and start period.I'm trying to tell with the start period, please try to execute these health check command only after the 10s. And if there is a failure, I'm telling please do the retries for ten times with an interval of 10s and inside the each check it has to wait for the maximum five seconds to get the response from this URL. So with this health check now my Docker or Docker   
compose is smart enough to identify if the service config server is started completely or not.

**Now, as a next step**, we need to define the dependency details of this config server into my accounts, loans and cards microservice. For the same under these accounts microservice just after the ports. Let us suppose it is not depends on config then it can be start before the config then sys will be carsh bcoz all the microservices is dependes on the config so it is mandatory first of all config microserivices will start properly with good health than other microservice should start .we can see the configuration in below code

loans:  
 image: "boss215/loans:s6"  
 container\_name: loans-ms  
 ports:  
 - "8090:8090"  
 depends\_on:  
 configserver:  
 condition: service\_healthy

it will try to do some checks to make sure that the service is started completely.

But if you don't provide this health check related information, it will never know how to perform the health checks that are specific to our config server.

That's why it is recommended to define your own health check and mention the condition as service healthy.

So with this my accounts microservice is going to wait till my config server started with a status as service healthy.

**As a next step**, we need to add one more service inside our docker compose. The service is related to rabbitmq.

We know our config server and all other microservices, they will depend on rabbitmq since we are using spring cloud bus inside the dependencies. So to make this work we need to create a new service element.Here to save some time. I'm going to paste few lines of code. Like you can see here

rabbit:  
 image: rabbitmq:4.0-management  
 hostname: rabbitmq  
 ports:  
 - "5672:5672"  
 - "15672:15672"

**host name** :- So this property is specific to Rabbitmq container. That's why we don't have to mention this for other services that we have here.  
  
**ports mapping :**  5672 and 15672.

We have done the same when we tried to run the Docker run command for Rabbitmq.Why two ports is, inside Rabbitmq there are two components.

One is which will take care of the management of the Rabbitmq and the other one which will take care of the core activities. So since there are two components, they are going to start at two different port like 5672 and 15672. That's why we need to mention these two port mappings.

Now, after this port mapping, we should also define the health check for Rabbitmq. Because my config server and other microservices, they are dependent on my rabbitmq. Until my rabbitmq starts completely with a good health.

I cannot really start my config server or any other service. That's why we need to mention this health check along with the test command. And at this test command I have given these value and these value is mentioned by the rabbitmq inside

their official documentation.

healthcheck:  
 test: rabbitmq-diagnostics check\_port\_connectivity  
 interval: 10s  
 timeout: 5s  
 retries: 10  
 start\_period: 5s  
networks:  
 - msbank

**rabbitmq-diagnostics check\_port\_connectivity** :- So whenever we want to test whether the Rabbitmq health is fine or not, we need to run this test command inside the container of Rabbitmq

After this, you know about this interval timer retries and start period.

Now we have the service with the name Rabbit.

**As a next step**, I need to define the dependency inside my config server.

To make depeneds on **rabbitmq**

depends\_on:  
 rabbit:  
 condition: service\_healthy

For the same, I'm going to invoke the element which is depends\_on and to this element I'm going to mention the service name which is Rabbit.

Since we want to make sure the rabbit is completely started successfully with a good health, we need to bring this to next line as an element under depends on and post that we need to mention colon followed by we need to mention condition. Under the condition we need to mention service healthy. With this now my config server will wait for my rabbitmq to get started and once it is get started, the health check that we have defined here should be successful. Then only the config server will try to start post that only my accounts, loans and cards will try to get started. And here you may have a question like our accounts, loans and cards microservice they also depend on this rabbitmq. Then why you are not mentioning this rabbit under the depends on of accounts, loans and cards.

It's very simple. The reason why I'm not mentioning is, anyway we have mentioned the dependency on config server inside the accounts, loans and cards.

When my config server is waiting for my rabbitmq to get started indirectly my accounts, loans and cards.They will also wait for this rabbitmq to get started.

That's why I don't have to mention again the rabbitmq dependency inside the accounts, loans and cards

We forget to mention these networks.

EasyBank under the Rabbitmq.

If you don't mention that, then rabbit service is going to start in a different isolated network and the connection between rabbit, config server and other microservices will not work. So that's why please make sure you are mentioning networks as EasyBank and the same network we have created towards the end.

Like you can see, we have created a network with the name EasyBank which supports a driver bridge. The same, we are trying to refer in all the services.

.

networks:  
 - msbank

Final compose file is shown below for your reference

services:  
 rabbit:  
 image: rabbitmq:4.0-management  
 hostname: rabbitmq  
 ports:  
 - "5672:5672"  
 - "15672:15672"  
 healthcheck:  
 test: rabbitmq-diagnostics check\_port\_connectivity  
 interval: 10s  
 timeout: 5s  
 retries: 10  
 start\_period: 5s  
 networks:  
 - msbank  
  
 accounts:  
 image: "boss215/accounts:s6"  
 container\_name: accounts-ms  
 ports:  
 - "8080:8080"  
 depends\_on:  
 configserver:  
 condition: service\_healthy  
 deploy:  
 resources:  
 limits:  
 memory: 700m  
 networks:  
 - msbank  
 environment:  
 SPPRING\_APPLICATION\_NAME: "accounts"  
 SPRIGN\_CONFIG\_IMPORT: "configserver:http://configserver:8071/"  
 SPRING\_PROFILE\_ACTIVE: default  
  
 configserver:  
 image: "boss215/configserver:s6"  
 container\_name: configserver-ms  
 ports:  
 - "8071:8071"  
 depends\_on:  
 rabbit:  
 condition: service\_healthy  
 healthcheck:  
 test: "curl --fail --silent localhost:8071/actuator/health/readiness | grep UP || exit 1"  
 interval: 10s  
 timeout: 5s  
 retries: 10  
 start\_period: 10s  
 deploy:  
 resources:  
 limits:  
 memory: 700m  
 networks:  
 - msbank  
 loans:  
 image: "boss215/loans:s6"  
 container\_name: loans-ms  
 ports:  
 - "8090:8090"  
 depends\_on:  
 configserver:  
 condition: service\_healthy  
 deploy:  
 resources:  
 limits:  
 memory: 700m  
 networks:  
 - msbank  
 environment:  
 SPPRING\_APPLICATION\_NAME: "loans"  
 SPRIGN\_CONFIG\_IMPORT: "configserver:http://configserver:8071/"  
 SPRING\_PROFILE\_ACTIVE: default  
  
 cards:  
 image: "boss215/cards:s6"  
 container\_name: cards-ms  
 ports:  
 - "9090:9090"  
 depends\_on:  
 configserver:  
 condition: service\_healthy  
 deploy:  
 resources:  
 limits:  
 memory: 700m  
 networks:  
 - msbank  
 environment:  
 SPPRING\_APPLICATION\_NAME: "cards"  
 SPRIGN\_CONFIG\_IMPORT: "configserver:http://configserver:8071/"  
 SPRING\_PROFILE\_ACTIVE: default  
  
networks:  
 msbank:  
 driver: "bridge"

**Now Optimizing Docker Compose File**

let's try to optimize our docker-compose.yaml file to not have any repetitive content for the same.

Inside the same folder where my **docker-compose.yaml** is present, I'm going to create a new YAML file. So under these default folder I'm going to create a new file and the file name I'm going to keep itas **common-config.yml**

Services:  
 network-deploy-service:  
 networks:  
 - msbank  
  
 microservice-base-config:  
 extends:  
 service: network-deploy-service  
 deploy:  
 resources:  
 limits:  
 memory: 700m  
  
 microservice-configserver-config:  
 extends:  
 service: microservice-base-config  
 environment:  
 SPRING\_PROFILES\_ACTIVE: default  
 SPRING\_CONFIG\_IMPORT: configserver:http://configserver:8071/"

**Here we**  will write common property inside service like we are using

networks:  
 - msbank

:   
many times so we create one service **network-deploy-service** and write this property so if we use this in another file then direct we will add **network-deploy-service** using file name and service name like below

extends:  
 file: common-config.yml  
 service: network-deploy-service

**Next Step (microservice-base-config)** :- same as we have deploy property many time in the docker.compose.yml

So for This dependency we are creating **microservice-base-config** Service to

re-use in docke-compose.yml using extend keyword with file name and service name like below

extends:  
 file: common-config.yml  
 service: microservice-base-config

**Next Step (microservice-configserver-config) :-** same as we have deploy property many time in the docker.compose.yml

So for This dependency we are creating **microservice-configserver-config** Service to re-use in docke-compose.yml using extend keyword with file name and service name like below

extends:  
 file: common-config.yml  
 service: microservice-configserver-config

**docker-compose.yml**

services:  
 rabbit:  
 image: rabbitmq:4.0-management  
 hostname: rabbitmq  
 ports:  
 - "5672:5672"  
 - "15672:15672"  
 healthcheck:  
 test: rabbitmq-diagnostics check\_port\_connectivity  
 interval: 10s  
 timeout: 5s  
 retries: 10  
 start\_period: 5s  
 extends:  
 file: common-config.yml  
 service: network-deploy-service  
  
 accounts:  
 image: "boss215/accounts:s6"  
 container\_name: accounts-ms  
 ports:  
 - "8080:8080"  
 depends\_on:  
 configserver:  
 condition: service\_healthy  
  
 environment:  
 SPPRING\_APPLICATION\_NAME: "accounts"  
 extends:  
 file: common-config.yml  
 service: microservice-configserver-config  
  
  
 configserver:  
 image: "boss215/configserver:s6"  
 container\_name: configserver-ms  
 ports:  
 - "8071:8071"  
 depends\_on:  
 rabbit:  
 condition: service\_healthy  
 healthcheck:  
 test: "curl --fail --silent localhost:8071/actuator/health/readiness | grep UP || exit 1"  
 interval: 10s  
 timeout: 5s  
 retries: 10  
 start\_period: 10s  
 extends:  
 file: common-config.yml  
 service: microservice-base-config  
  
 loans:  
 image: "boss215/loans:s6"  
 container\_name: loans-ms  
 ports:  
 - "8090:8090"  
 depends\_on:  
 configserver:  
 condition: service\_healthy  
  
 environment:  
 SPPRING\_APPLICATION\_NAME: "loans"  
 extends:  
 file: common-config.yml  
 service: microservice-configserver-config  
  
 cards:  
 image: "boss215/cards:s6"  
 container\_name: cards-ms  
 ports:  
 - "9090:9090"  
 depends\_on:  
 configserver:  
 condition: service\_healthy  
  
 environment:  
 SPPRING\_APPLICATION\_NAME: "cards"  
 extends:  
 file: common-config.yml  
 service: microservice-configserver-config  
  
networks:  
 msbank:  
 driver: "bridge"

**Generating docker image and pushing them into Docker Hub**

**Generating Docker Image :-**

Open microservices location in cmd and write below cmd :-  
**E:\Microservices\section6\v2-spring-cloud-config\loans> mvn compile jib:dockerBuild**

**E:\Microservices\section6\v2-spring-cloud-config\accounts > mvn compile jib:dockerBuild**

**E:\Microservices\section6\v2-spring-cloud-config\cards > mvn compile jib:dockerBuild**

**ConfigServer :-** in config server microservice we don’t have a jib dependency so we will paste this below dependency and

<plugin>  
 <groupId>com.google.cloud.tools</groupId>  
 <artifactId>jib-maven-plugin</artifactId>  
 <version>3.3.2</version> *<!-- Check for the latest version -->* <configuration>  
 <to>  
 <image>boss215/${project.artifactId}:s6</image>  
 </to>  
 </configuration>  
</plugin>

run the cmd to generate the docker image

**…../section6\v2-spring-cloud-config\configServer > mvn compile jib:dockerBuild**

**Pushing Image into the docker Hub**

**Cmd :- PS E:\Microservices\section6\v2-spring-cloud-config\configServer> docker image push docker.io/boss215/accounts:s6**Since I'm trying to push into the Docker hub followed by what is my Docker account name, which is **boss215** and what is the image name like **accounts:S6**. So this image should also present inside your local system then only this push will work.Please make sure you login into your Docker desktop and your docker is running whenever you trying to run this command, so I'm trying to push into the Docker hub. After a few seconds, my account's Docker image is successfully pushed into the Docker hub.

**Pushing loans :- …/configServer> docker image push docker.io/boss215/accounts:s6**

**Pushing cards :- …/configServer> docker image push docker.io/boss215/cards:s6**

**Pushingconfigserver :-/configServer> docker image push docker.io/boss215/configserver:s6**

Now we can see all the images are pushed into the docker hub **https://hub.docker.com/**

**Testing Config Server Changes end to end using docker compose & default profile**

let's try to use the Docker compose file to start all our containers and make sure the configuration related changes are working even inside the Docker environment. Before we try to run the Docker compose commands, please make sure there are no running containers. Inside my Docker desktop,

.If any containers running then stopped containers, please delete them because you need a lot of space and memory inside your local system. To run the five different containers, including like rabbit,

**config server**, **accounts**, **loans** and **cards**. So I came to my terminal.

Here I'll go back to location where my Docker compose folder is present, so I'll navigate to the Docker compose folder. So inside the Docker compose folder, as of now we have created the Docker compose file only inside the default folder. So let me go inside this folder.

Here I'm going to run the command, which is docker compose up -d. So as soon as I try to execute this command,

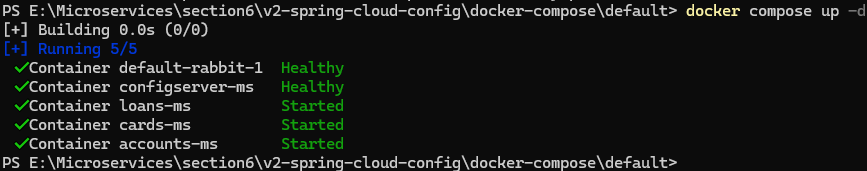
**PS E:\Microservices\section6\v2-spring-cloud-config\docker-compose\default> docker compose up -d**

We can see the containers are trying to get started. As of now,  
so first my **rabbitmq** service will try to get started. Post that my **config server** will try to get started once my config server is created, then only my **accounts**, **cards** and **loans** related containers will get created. So it will take some time. So please wait for some time here.

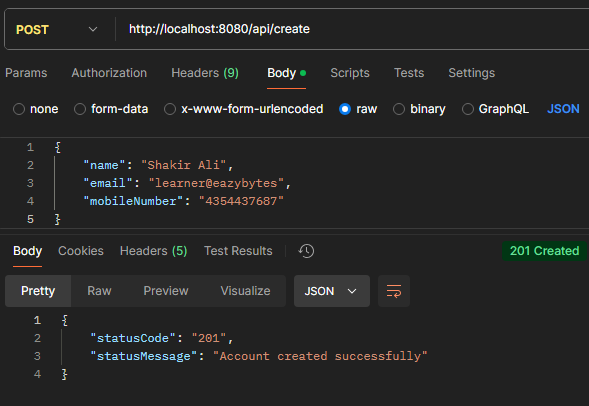
Meanwhile, you can open a new terminal and try to run the command which is docker ps. So this will show you the list of running containers. So as of now you can see inside my local all my containers started. Very first one is rabbitmq and the status is healthy because we have provided the health check very similarly config server, it has started and the status is healthy because we have provided the health check details. Whereas accounts, loans and cards, they simply started less than a second ago and my docker is not sure about their health status, because we didn't provide any health instructions on how to check that.

But that's fine. Whenever the config server and rabbitmq started our accounts, loans and cards, microservice also should get started automatically.

**We can see sequence of our services it is running according to our yml file**



**Now we can see in docker dashboard as well as try to run using Postman our micrservices running properly**



**If we change docker to dck in github account.yml then want to fetch the info from the postman then we will get the error inside the terminal where hookdeck is open we are getting 500 error that means reresh attempt failed . now check log inside the config server container here we can see it is trying to connect our rabbitmq with the default connection that is localhost:5672 so this is not going to work bcoz rabbitmq not started within the same container of configserver it has started as a separate container with a separate service name so we missed the rabbitmq configuaration details inside the docker-compose file**

may have a question like how come other functionality is working

like we tested accounts microservice, APIs and we are getting a successful response, so let's try to check the same.

So if I try to execute this actuator/health, you can see I'm getting the status as down, which means the overall health is down.

[**http://localhost:8071/actuator/health**](http://localhost:8071/actuator/health)

Whereas if I try to check the readiness **http://localhost:8071/actuator/health/readiness**

I'm getting the status as up, because according to the config server rabbitmq is an optional settings.(means if rabbitmq have some issue then also our server will be in ready state readiness)

So to fix this issue, it's very simple.

We need to provide the rabbitmq connection details for all our containers.

So let me go to that Docker compose file. Inside the docker compose file, we can define the connection details under each container. Instead of that, we can also define the connection details under the microservice based config, because

my config server accounts, loans and cards microservice, all of them needed the rabbitmq related configurations. That's why I'm going to create a new element under the deploy with the name environment. Under this environment I need to mention the properties related to the Rabbitmq. So if you see here as of now inside the **application.yml of serverConfig**

rabbitmq:  
 host: "localhost"  
 port: 5672  
 username: "guest"  
 password: "guest"

we have mentioned these properties like

which is spring.rabbitmq host as localhost port and username and password.

So the port username and password we should be fine. So these are the default values and my docker containers also they can use that, but this localhost

will not work. Instead we need to mention the **service name** of Rabbitmq.

So that's why let's try to override only these host variable. For the same inside the **common-config.yml**

microservice-base-config:  
 extends:  
 service: network-deploy-service  
 deploy:  
 resources:  
 limits:  
 memory: 700m  
 environment:  
 SPRING\_RABBITMQ\_HOST: "rabbit"

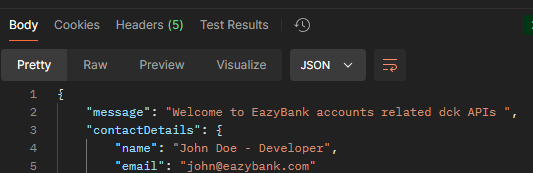
in upper image we create a new environment variable with the name **spring\_rabbitmq\_host.** The host name, we should not mention localhost.

Instead we should mention the service name which is rabbit. So all my containers they can connect with this service because they all started inside the same network. So let me mention the host as rabbit here. Maybe I can try to mention inside the double quotes. So with this the issue should get resolved.

Now all the things is fix we can see inside the configserver container logs

Now we can check our actuator health and readiness is up

Now I have change docker to **dck** inside the github properties file and hinting api in postman and able to get updated values



**Preparing Docker Compose File For QA and prod Profile**

As of now, we have this Docker compose file created for the default profile under the default folder.

We want to create very similar kind of Docker compose file for prod and qa profile. So that based upon your requirements you can always try to run the corresponding docker compose Yaml file before I try to create the new Docker compose files. First, let me stop all the running containers.

So here inside the terminal where I have started the Docker compose up command, I'm trying to clear

the console post that I will try to run the command docker compose down this will stop and delete

all the containers inside my system. Now I'll go to my IDE. Here creating Docker compose for the new profiles is very, very easy, so let me show you how easy it is.I'm going to select these two files from the default and paste the same under the prod.After this prod, I'm going to paste the same inside the QA as well, so let me paste them.You can see now we have the same files inside the QA and prod.

Do you know how many changes I have to do inside these docker compose files that I have created under

Prod and qa. There is only one change that we have to do which is under the commonconfig.yml instead of this default. I need to mention prod here and similarly I need to open the commonconfig.yml and change this spring profiles active as QA. This way with only one environment variable.

We are trying to control the behavior of our containers. Externally, we are using the same docker image to run the containers in qa environment, prod environment and local environment. So this is the power of Docker images and Docker containers. Along with the spring boot framework, we are able to use the same immutable Docker image across all the environments. And if you have any specific requirements based upon your environment for suppose like inside the prod environment, if you want to give more space for your containers, you can mention that with the help of commonconfig.yml instead of 700 mb,

you can give more memory. This way we have the freedom also to maintain different, different requirements and configurations for various environments. Now we have two different Yaml files available for prod and qa profile as well. Let's try to test one of them and make sure everything is working. So which one you want me to test here? So tell me whether you want to go with prod or qa, you want to go with prod. I know you like always production, so let me go with the prod. For the same inside my terminal, I need to navigate to the folder which is prod. And inside this prod I need to run the command which is docker. compose up. So let me run this command. This will start all my containers and my webhook is running inside the other terminal.I didn't disturb anything here, so let's wait for our containers to get started. Now all my five containers are started. I can also confirm the same inside the Docker desktop by opening this prod profile and clicking on this rabbitmq you can see the Rabbitmq started successfully. Now let me check the config server. Config server also started successfully.

Now I can open the loans microservice.It also started successfully the same I can confirm for cards.

This also started successfully. Now I'll check for accounts microservice and this one also started successfully.Now let's try to test the changes.This time I'll try to change the property related to the cards microservice and see if it is going to work.Ffor the same, I'm going to my eazybytes-config and here I need to change the values inside the cards-prod.yml.Because right now we started the microservice containers with the prod profile. So let me click on this and post that, I can click on this edit button. Here instead of prod, I'm going to mention that Docker APIs. Now let me commit these changes to the GitHub repo. Once I have committed these changes, I'll go directly to my postman. Inside my postman, I need to invoke the cards microservice contact-info API. So I'm trying to invoke this. You can see we are getting Docker APIs as an output, so let me try to revert the changes and test one more time so that we'll go back to the original message value.Here I'm clicking on this edit button and replacing this docker with the prod and committing the changes to the GitHub repo. I'll go to my postman, this time I'm trying to click on the send button again. You can see I'm getting the prod APIs, so there might be a delay in refreshing like wait for five to 10s. Post that you should be able to see the updated property inside your postman response.

With this we have successfully tested all the configuration management related changes both locally

and using the Docker compose, so I'm going to check in all the changes present inside this folder into the GitHub repo. Please refer the same for any questions or for any issues. We discussed a lot about configuration management

**Using MySQL DB inside microservices**

I'm going to fix our code to point to the real database like MySQL database.

As of now, we are using H2 database and this internal H2 database is not recommended for real projects or production applications. That's why inside this section I'm going to focus on how to use MySQL as a backend for our microservices. As of now we have three different microservices like accounts, loans and cards. I'll make code changes so that these microservices will utilize MySQL database.

With that quick introduction,

We are copying all the files from the previous project which was section 6

So let me take all these projects and folders.

After copying this and going back to my Section 7 and I'm going to paste the projects and folders

that we have created. So let me make sure there are no hidden files related to idea. So there are no hidden files here.

As a next step, I'm going to open this Section 7 related code inside my IntelliJ idea.

So to make sure they are getting loaded as Maven project, I need to make sure I'm clicking on this

load button. So with that all my projects might have loaded as maven projects.

So Docker Compose is not a Maven project.

It is a simple folder where we are trying to store all the Docker compose files.

For now let's not worry about it. As a next step before I try to make some MySQL related changes. I want to delete all the changes that we have done regarding Spring cloudbus and spring cloud config monitor.

The reason I don't want to carry the spring cloud bus and config monitor related changes inside our config server and other microservices is, I want to get rid of rabbitmq container as a dependency inside our microservices.

There is no harm you carrying that dependency. But since we are trying to learn by building these projects and running these projects inside local system or running many containers and many dependencies inside your local system may not be a good idea. Because there are more containers, more services we are going to onboard in the coming sections. And if I try to start them, all of them, with the help of Docker compose or inside my local host, it is going to make my system slow. And I face some issues like when I'm trying to use all these containers inside system,

who are going to use 8 GB Ram laptops, so their laptop is going to be slower, even more compared to 16 GB Ram. That's why since we are already clear about spring Cloud bus and Spring Cloud Monitor, we don't need

to carry that rabbitmq dependency to all the future sections that we are going to discuss.

So to remove the spring cloud config bus and config monitor related changes.

First let's go to the config server pom.xml.Here look for the dependencies with the name spring-cloud-starter-bus-amqp and spring-cloud-config-monster.

So let's remove both of them with that.

*<!-- https://mvnrepository.com/artifact/org.springframework.cloud/spring-cloud-starter-bus-amqp -->*<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-bus-amqp</artifactId>  
 <version>4.2.0</version>  
</dependency>  
<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-config-monitor</artifactId>  
 <version>4.1.1</version>  
</dependency>

Now I'll go to the application.yml file of config server. So here I'm going to remove certain properties like you can scroll down there are rabbitmq related connection details which you have mentioned.

There is no need of them, we can remove them. And apart from that we also have actuator management related properties. We can keep these properties because if needed we can refresh our configurations by manually invoking the actuator refresh API against each instance.

And on top of that anyway, we need these readiness and liveness probe related configurations, so we should not be deleting them

rabbitmq:  
 host: "localhost"  
 port: 5672  
 username: "guest"  
 password: "guest"

. **Now I'll go** to the accounts microservice pom.xml and I'll delete the dependency related to the spring-cloud-starter-bus-amqp

*<!-- https://mvnrepository.com/artifact/org.springframework.cloud/spring-cloud-starter-bus-amqp -->*<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-bus-amqp</artifactId>  
 <version>4.2.0</version>  
</dependency>

After this I'm going to load the maven changes

**as a next step**, I'll go to my application.yml of accounts microservice. So here also I'm going to delete the properties related to the rabbitmq. Here we have rabbitmq properties so let me delete them.

rabbitmq:  
 host: "localhost"  
 port: 5672  
 username: "guest"  
 password: "guest"

Now as a next step I'll do the similar changes inside the **cards** and **loans**

So now with these changes we don't need to run the rabbitmq container inside our local system because we don't want the automatic refresh of the properties with the help of bus refresh or with the help of GitHub webhooks process that we have discussed.

**So as a next step**, now if I want to delete the H2 database inside my microservices first I need to have a running MySQL database inside my local system. If you are a traditional developer or if you are working in a monolithic application, the obvious approach is you will install a MySQL server inside your local system by following all the steps and post that you will try to create that database. Once you have created the database, you will start the server and after starting the server only you can start integrating into the microservice application. But now we are **smart developers.**

We are **microservices developer** who know Docker. Like I said, one of the primary advantage of Docker is, we don't have to install any component or software inside our system. With the help of Docker images, we can run any component or any dependency inside our local system as a container. And once we are done using with that software component, you can simply stop the container or delete the docker container associated to it.

So that's why now I don't recommend you to installing the MySQL server manually by downloading it because it is going to take a lot of space inside your system.

Then how to have a running MySQL database for my three different microservices, like I said, the best practice is to have a separate database for each of the microservice.

For the same, I'm going to show you how to create a local running MySQL database inside your system with the help of Docker.

So please make sure you have started the Docker because I'm going to run a Docker command, for the same we need the Docker to be running inside our local system

**As a next step**, I'll open the terminal. Inside the terminal, I'm going to run the command,

**docker run -p 3306:3306 --name accountsdb -e MYSQL\_ROOT\_PASSWORD=root -e MYSQL\_DATABASE=accountsdb -d mysql**

which is Docker Run and I'm going to provide the port mapping. We know, like by default, the MySQL database is going to start at the port 3306.

**-p 3306:3306:-** So the same port I want to expose to the local microservice, which is running inside my system. When I try to create a Docker container like we know it is going to run inside its own network in order to get that expose to our local network.

(if mysql not running in our sys then this will downloand the mysql image from the dockerHub)

Inside our system, we need to make sure we are exposing that with the port number like 3306. After this port mapping,

**--name accountsdb :-**  I'm going to provide a name to this database that we are going to create. The name that I want to keep it here is, accounts DB because we are going to create two more DBS like loans DB and cards DB. So to differentiate between the DBS, I want to give a name here, and the name that I'm trying to give is accountsDB.

Now whenever we are trying to use a MySQL Docker image to create a MySQL Docker container, we need to provide certain properties like what is the root password that you want the Docker to be considered while creating the container.

**accountsdb -e MYSQL\_ROOT\_PASSWORD=root :-** So that's why by using -e, so -e indicates that I'm trying to provide an environment variable to this Docker run command. So here I'm going to mention MySQL root\_password. So to this environment variable, I'm giving the value as root, which means I want the password for the root user as a root itself. So by default the root username is going to be root.

I don't want to change that, but since we want to define our own password, I'm providing this environment variable. After this, I'm going to provide one more environment variable and this environment variable name is MySQL database

**root -e MYSQL\_DATABASE=accountsdb :-** . So like we know whenever we install a MySQL database server, you will get an empty server by default while there won't be any database or schema installed inside that database server. So that's why I want to create an my own database or schema inside this MySQL Docker container that we are going to create.

So by the time my Docker container of MySQL is ready inside that I'm going to have an database with the name accounts DB. So that's why we need to mention the same. After mentioning these environment variable, I'm going to mention a flag which is **-d. -d** indicates I want to start this command in a detached mode and at last we need to mention what is the Docker image name of MySQL. So the Docker image name of MySQL is MySQL itself. So I hope you are clear with this command. So let me execute this command. You can see I got a containerId as an output.

**C:\Users\User>docker run -p 3306:3306 --name accountsdb -e MYSQL\_ROOT\_PASSWORD=root -e MYSQL\_DATABASE=accountsdb -d mysql**

**845872dd21d8560d6b34a82e56b8bedb5e4be47924068c02f54b87dc93d9b601**

Now if I go and check my Docker desktop, I will be able to see a running container inside my local system with the name accounts DB.

So this using MySQL Docker image and it exposes the traffic to the outside with the port 3306. Now I can connect to this database and see if my database is properly created. So to connect to this database, we need some client.

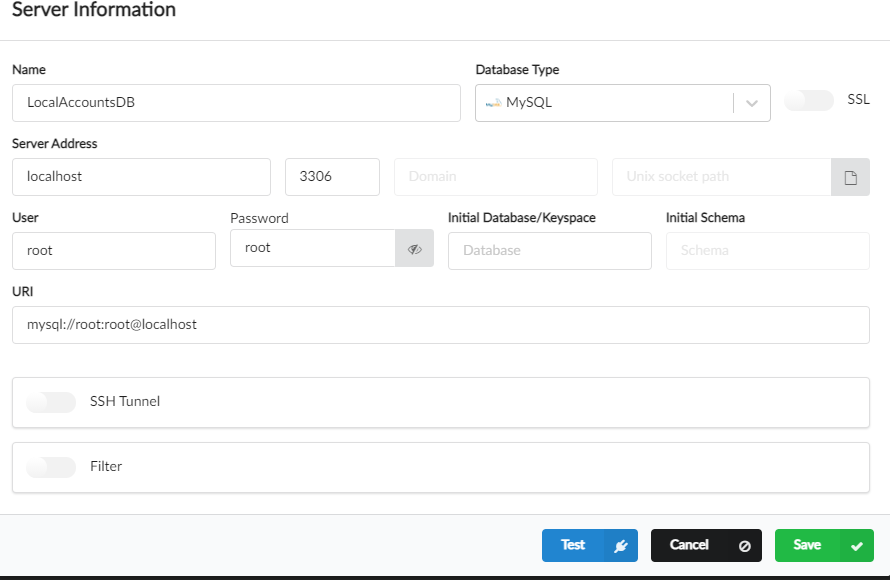
So I'm going to show you here a simple client that you can use inside your laptop.

There is a client with the name SqlElectron, so this is going to be a super, super lightweight application inside your system.

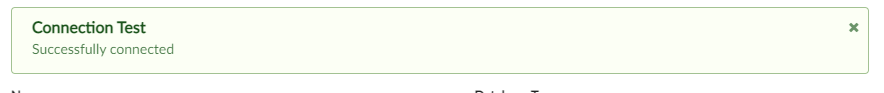
Step 1 To Download SqlElectron :- Go to <https://sqlectron.github.io/>

Click on Download GUI and download the exe file(<https://github.com/sqlectron/sqlectron-gui/releases/tag/v1.38.0>) and install it

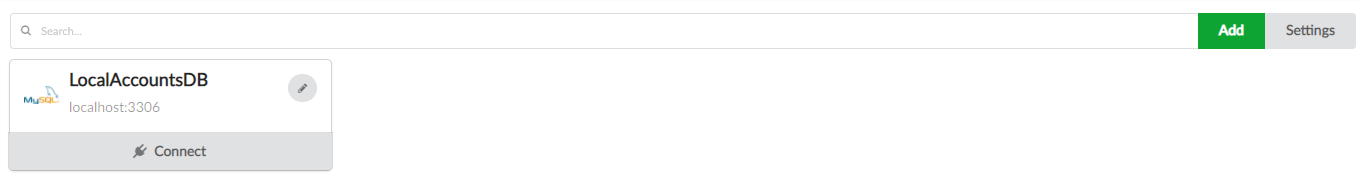
Step 2 :- create a new connection -> click on add Btn u will get this popup fill like below



Now click on test btn then we will get below response

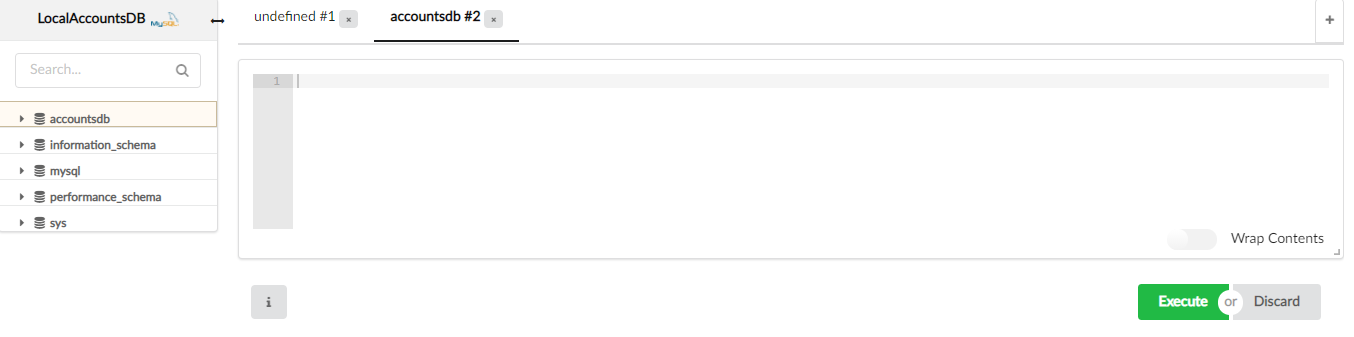


After that saved it after saved if will get below response



After that click on connect btn we can see our db is connect successfully we have accoutdb which is created inside the docker container mysql database server

As of now this database is empty does not have any table



We're going to create those tables, don't worry.

So now I have a MySQL database ready inside my local system that supports my accounts microservice. With a single command, I'm able to achieve this. It didn't even take 10s. Whereas if you compare with the traditional approach, it is going to take a lot of time and a lot of space inside your local system. In real projects we usually don't install a MySQL inside our local system. The project infra team, they will deploy into a dev server or QA server and they'll ask us to use the same inside our local code. But since right now we are trying to learn the microservice everything on our own. So I want to show you a way on how you can create these MySQL databases very easily with the help of Docker. So that's what I'm trying to achieve here. So now my accounts microservice database is ready.

**As a next step**, I want to create database for cards microservice and loans

Microservices .so we will follow the same process for these microservices.

**Cmd for loan**

**docker run -p 3306:3306 --name loansdb -e MYSQL\_ROOT\_PASSWORD=root -e MYSQL\_DATABASE=loansdb -d mysql**

Now if I try to run this command, I'll get an error.

Let's see what is the error. You can see it failed to start because inside my local system, 3306 port is already being used by the accounts database. So that's why I should expose the loans database into my local system with a different port number. So let's try to change my port number here

**docker run -p 3307:3306 --name loansdb -e MYSQL\_ROOT\_PASSWORD=root -e MYSQL\_DATABASE=loansdb -d mysql**

The port number I'm going to mention here is 3307. So now if I try to start this container, it is throwing me an error saying that container name already exists. It seems the previous command created a container, but it never exposed the traffic to the outside world because 3306 is already being used. So what we can do here is we can delete this container which is created with the previous command. So let me delete this. Now I can execute my Docker command.

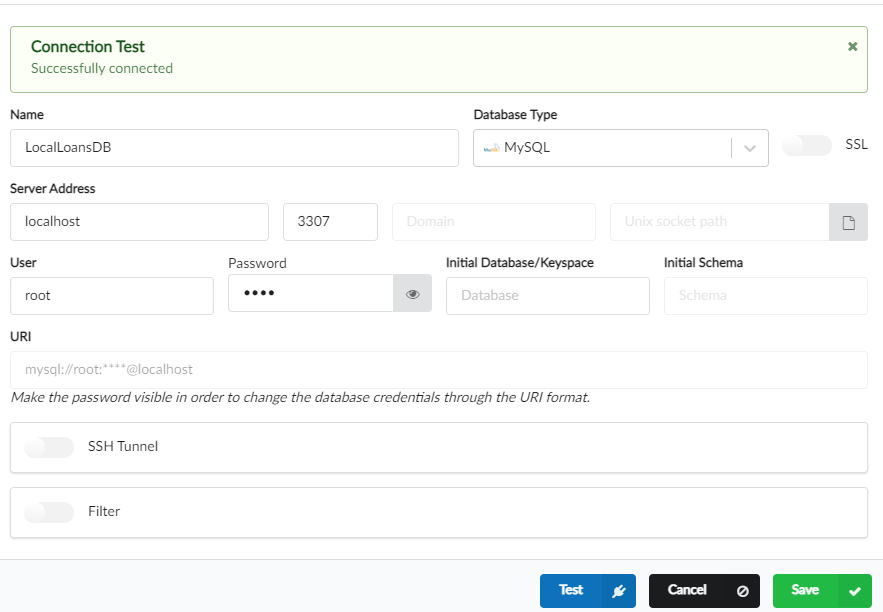
So the command this time is going to be successful. Like you can see, I'm getting the container ID as an output. With this, you may have some question like I'm using the same 3306 for loans and accounts microservice. Then why Docker is not complaining? Like you said, each container will have its own ecosystem, will have its own isolated network. That's why since these loans DB and accounts DB They are two different containers. It is going to work perfectly. Whereas when we try to expose to the outside world, the outside world is our local system. Inside the local system, 3306 we are already using for accounts DB and we cannot use the same file loans DB. That's why we need to correct that only the first port number and we don't have to correct the second port number, which is Docker internal port number. I hope you are clear with this discussion.

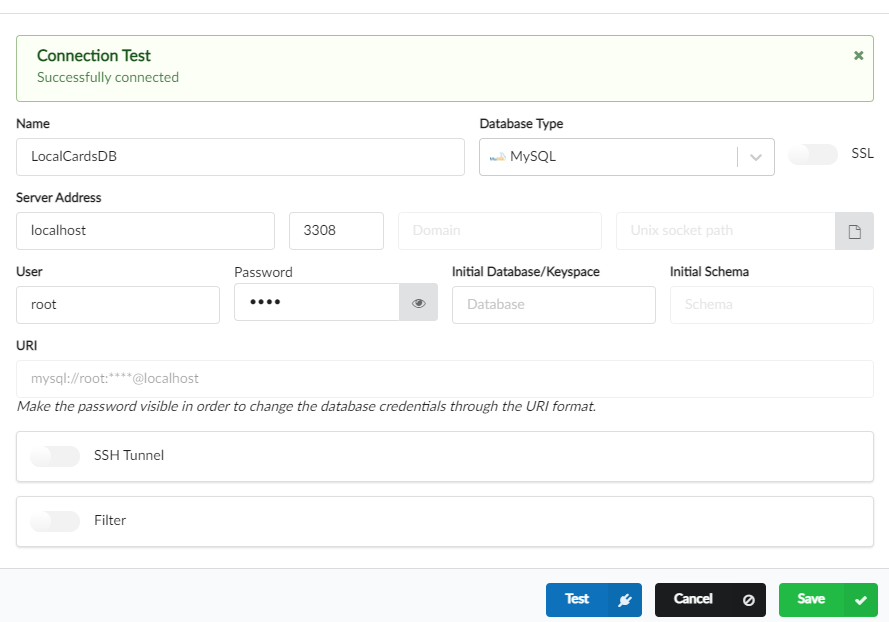
**cmd for Cards**

**docker run -p 3308:3306 --name cardsdb -e MYSQL\_ROOT\_PASSWORD=root -e MYSQL\_DATABASE=cardsdb -d mysql**

now connect loans and cards connection inside our sqlElectron for this we will follow the step as accountsdb…

**Step 1** :- logout from the electron then we can see add btn and will follow same step same as accountsdb





**Update microservices code to replace H2 DB with MySQL DB**

**Step 1** :- Remove the h2 database dependency and add below dependency in all of three microservices, loans, cards, accounts

<dependency>  
 <groupId>com.mysql</groupId>  
 <artifactId>mysql-connector-j</artifactId>  
 <scope>runtime</scope>  
</dependency>

**Step 2** :- Remove h2 database configuration from the application.yml of all the microservics and write mysql configuration inside the accouts, cards, loans applicatins.yml file

datasource:  
 url: jdbc:mysql://localhost:3308/loansdb  
 username: root  
 password: root  
jpa:  
 show-sql: true  
sql:  
 init:  
 mode: always

In the H2 database during the startup, my spring boot automatically look for the SQL file with

the name schema.sql and execute all the tables that we have mentioned inside this SQL file.

But with the MySQL database or any other real database, the spring boot will not follow the same behavior. It will expect the developers to have the completely created database available by the time the spring boot application is trying to start. But at the same time here, we don't want these scripts or tables to be executed manually. Whenever I'm trying to create a new container or a new database.

So to overcome this challenge, there is a property that we can mention. The property is, under the JPA I'm going to mention a new element

**sql:  
 init:  
 mode: always**

which is SQL under the SQL init, after the init I'm going to mention the element mode and to this mode I'm going to pass a value which is always. So with this I'm telling to my spring boot.

Please always try to execute the scripts that I have mentioned inside the schema.sql. You can see here

*CREATE TABLE* IF *NOT EXISTS* `loans`

I'm trying to use a script which is if not exist, so my spring boot framework will only create the table if it is not available inside my MySQL database. Maybe during the very first time the table will be created and from next time onwards, if I'm using the same Docker container, my spring boot is not going to create these tables because I have mentioned these If not exist. If you don't have this, if not exist, then you are going to get an error during the startup that it tried to create the tables, but they are already present.So to avoid that, please make sure your script has this if not exist.So now we have made all the changes inside the MySQL database, so let me save these changes as a next

step.

**Step 3**:- I'll try to start the config server because our micro service has dependency on the config server. That's why I'm going to first trying to start the config server once the config server is started successfully. So here I'm trying to start the accounts application. Now let me go to the cards application and try to start the same inside the debug mode after the carts application. Obviously the next microservice that we need to start is loans microservice. Here I'm trying to start my loans application in a debug mode. Now all my microservices are started successfully.First, let me go and check how the database is looking like. Here I have three different connections. First, let me try to connect to the accounts database. Now, under this accounts database, our accounts DB schema, we should have that tables created during the startup. So let me open this. We can see tables are created.

**Step 4** :- So let me test a few scenarios inside my postman. So here, inside my postman under Microservices, I'm going to open the account. And inside this we have create account. So I'm trying to create a new account with this mobile number. So let me submit this request and getting a successful response and using the same mobile number, I'm going to create a new card by using the createCard API.

So I'm trying to submit this request. A new card is created. I'll also do the same for loans microservice. So here I'm trying to click the Create loan and this one also, you can see we are getting a successful response.Let's try to validate if these changes are saved into the database.As of now, I'm inside the cards database, so let me try to click on this cards and you can see these are the card details that we have submitted and all of them are getting saved here. As a next step, I can confirm the same for loans database. Also inside the loans database, we have loans table and if I try to run this, you can see we have

the data saved into the database. Now, let me confirm for accounts database and the accounts database also, we have two tables. First is customer where the customer details are stored followed by account details. So let me confirm the accounts details also. So the account also created successfully. This confirms our microservices are able to talk with the local running database containers and they're

able to save the data. Since right now we are using the Docker containers. Whenever you delete this container, all the data that you saved inside this particular database will be lost forever because this works very similar to local installation. Whenever you install your MySQL inside your local system and stored data, you will be able to see it. But someday, if you try to uninstall your MySQL installation, then all the data will be lost. Very similarly, you need to make sure these containers you're not deleting them. You can stop them but don't delete them. If you delete them, you will lose the data. So that's a catch here.I can also show a demo if you want,for example, first let me stop all the running microservice.

So I'm trying to stop accounts application and similarly I'm trying to stop loans application followed by cards application.

Now here I'm going to stop the accounts database loans, database and cards database. As of now, all my containers are in stopped status. Now, if I try to start my microservices, definitely they will throw an error that they are not able to connect to my container to show you a demo.What I can do here is I can try to delete the cards database container.So let me hard delete this. And this means I'm losing this container forever. Whereas the other two containers I will try to restart them. Like right now they are in stop status.

I'm trying to start them, but we don't have cards database. We need to create it again.

So let me go to the terminal. Here I'm trying to run the command related to cards database and it is going to expose at the port 3308. So let me execute this and this will create a new container with the name cards database. And now all of them are running.The output that we can expect here is, all the data related to accounts, database loans database should be there, whereas cards database data will not be there.

So I can also show you that physically inside the SQL Electron. So let me log out first let me show you the cards database and connect to the cards database. You can see there are no tables itself. The data, the tables, all that we have created will be lost because we had deleted the container. Whereas if you go and check any of the accounts or loans database, you should be able to see that data still because you just restarted the containers. The data that you stored inside the container is never lost. As long as you are maintaining that inside your Docker server, even if you restart your laptop or Docker server, the data and the container will not be lost. So now let's try to start our microservices. So here I'm trying to start accounts application followed by cards application and at last I'm trying to start loans application. So let me go to the Postman Inside the Postman. This time I'm going to run only the fetch API. And at the accounts there is a fetch account details. So let me open that and try to send the same mobile number and you should get the response because the data is stored inside the database. I can run this fetch card details. This time I will not get any output because all the data that we saved is lost because we had deleted the container, whereas with loans it should work perfectly. You can see we are getting the output. I hope you are seeing the drawback that we have whenever we are using the Docker containers inside our local system. It has both advantages and disadvantages. You can always spin up the database very quickly, but we need to make sure the container is never deleted. Even if you delete by mistake, you are going to lose the data forever.So please have this information and accordingly use the docker MySQL containers.

I hope you are clear.

**“In real production, MySql DBAs will attach a storage or volume where the data can be stored by the MYSQL container. That’s Why in real prod MySQL containers, we never lost data even if we delete or replace the MYSQL container”**

**Update docker compose file to create and use MySQL DB**

As of now, inside our local system, our microservices, they are able to connect with the MySQL database

containers that we have created. But when you try to run your microservices in a container environment like Docker Compose or Kubernetes, whatever properties we have mentioned inside the application.yml, they are not going to work. Because we have hardcoded the hostname as localhost. That's why whenever we are using Docker compose file or whenever we are trying to deploy our microservices in a Kubernetes environment, we should provide these data source URL, username and password using environment variables. Inside this lecture I'm going to update our Docker compose files and using the Docker compose file, we are going to start all our database containers along with the microservices and we'll establish a communication link between them so that they can communicate with each other. For the same, first, I'll try to stop all my running microservices instances. Along with that, I'm also going to stop all my running containers and delete them because we need to create these containers with the help of Docker compose, but not by running the Docker commands manually. So let me delete them. Now, I don't have any containers inside my system. As a next step before I try to change the Docker compose file.

First, we need to regenerate the Docker images based upon the latest changes that **we have done inside our microservices. We have removed the h2 dependencies and we have added the MySQL dependencies and the properties. That's why we need to regenerate the images based upon the changes that we have done inside this section**. So first let me go to the pom.xml and here I'll change the tag name from S6 to S7 I'll do the same for all type of applications that we have, like inside cards also, I'm going to open the pom.xml. I will rename the tag name from S6 to S7 and refresh the maven changes. Now let me do the same for config server. Technically we don't have to regenerate the docker image for config server, but to be consistent, I'm trying to mention the same tag name even inside the config server. So now let me do the same for loans as well. Inside the loans I'm going to update the tag name from S6 to S7. So with this all our pom.xml are in good shape.

<configuration>  
 <to>  
 <image>boss215/${project.artifactId}:s7</image>  
 </to>  
</configuration>

**As a next step**, I need to go to the terminal and generate the Docker image.

I have opened a new terminal inside the section 7 folder. Here I need to go inside the accounts microservice. So to generate the docker image for accounts microservice I'm going to run the command docker compile **jib:dockerBuild**

**E:\Microservices\section7\accounts>mvn compile jib:dockerBuild**

So before I try to run this command, let me make sure to delete all the existing Docker images inside my local system. So inside my desktop, I'm going to the images and I look for all the section 6 related images. So I'm trying to delete them just to get some space inside my local system. So please regularly delete unused images, unused containers inside your local system also. So that you get new storage and new memory inside your local system.

And with that, your system will work fine even if you try to start multiple containers with the Docker compose command. So I'm trying to run this command. So this will generate a new Docker image for accounts microservice. So the Docker image is generated successfully.

And same as create the docker images for **card**,**loans** ,

**As a last step**, **let me generate the Docker image for config server. So let me go inside the config server and from the config server folder I'm going to run the Maven, compile jib dockerBuild command. With this, we should have all the Docker images inside our local system. You can confirm the same by looking the images here inside the Docker desktop. Later on I will push them into my Docker hub repository. I would also recommend you to push your own Docker hub repository so that they are stored inside a centralized repository. So now we need to update the Docker compose file. For the same, I'm opening this docker compose. First, let's try to update the Docker compose file present inside the default folder. So let me open this, here first we no more needed these rabbit related service because right now we have reverted all the spring cloud base related changes. So let me remove this and instead of this service, I'm going to create three different services, one for accounts database, and second one for loans database and third one for the cards database**

accontsdb:  
 image: mysql  
 container\_name: accountsdb  
 ports:  
 - "3306:3306"  
 healthcheck:  
 test: ["CMD","mysqladmin", "ping" "-h", "localhost"]  
 interval: 10s  
 timeout: 5s  
 retries: 10  
 start\_period: 10s  
 environment: :  
 MYSQL\_ROOT\_PASSWORD: "root"  
 MYSQL\_DATABASE: "accountsdb"  
 MYSQL\_USER: "root"  
 MYSQL\_PASSWORD: "root"  
 extends:  
 file: common-config.yml  
 service: microservice-base-config

So the service name, I'm going to keep it as accounts DB. To this service, we want to use the image which is MySQL itself and the container name we can use as accounts DB itself. And after the container name, we need to mention the ports mapping for the same I need to invoke the

**Ports :-**  and here for the accounts database we need to mention that 3306 and 3306 after the port mapping, I'm going to mention the

**health check :-** details and how to test the health of these accounts database.

Once these accounts database is completely started, then only I should try to start my accounts microservice.

Since we have that kind of dependency, we need to define health check details so that my docker compose can try to start my accounts microservice only after accounts database becomes healthy. And the

**Test** command that we need to mention here is, inside the square brackets I need to give these values with these command I'm trying to tell to my docker, please run this ping command where it is trying to ping the host with the value localhost. Whenever it gets a successful response.

That means the health of this container is healthy. You can get all these health check commands from the official documentation. You can always look inside the web like what is a health check command for the MySQL docker container? Similarly, if you are using some other Docker image like for Oracle or Redis, you can always look for these health check related commands inside the official documentation. So after this test command,

I'll try to mention

**timeout as 10s**. So it has to wait only 10s. Within the 10s, if it is not getting the response for the ping, it will consider it as failure.

And I also want to mention retries.

**So retries** I will mention as 10 times and apart from this timeout and retries, we can also mention this interval and start period. I can say

**Interval** as 10s and similarly I can mention

**start period** as 10s. So with this we have defined the health check details.

After the health check details, I also want to provide some environment variables for my accounts database container. And these environment variables are like MySQL root password and MySQL database. We pass the similar set of environment variables when we are trying to create the container with the help of Docker run command.So the same I'm trying to mention. Now we are creating a container,

**as a next step** we need to make sure that this container is tagged to the same network where the other containers are trying to start. Then only they can communicate with each other with the help of service name. Otherwise, the communication between our microservices and the database containers will not happen. For the same, what we can do is, we can take this

extends:  
 file: common-config.yml  
 service: microservice-base-config

extends configuration from the config and I'm going to mention the same inside the accounts database.

. So let me copy this value post that, I'll go to the Docker compose file and I'll mention the same here. So with this now my accounts database is ready. Very similarly, I need to create loans, database and cards database.

So with this we should be good, but we need to establish a link between our microservices and with

these database containers. For the same, if you go to the common-config.yaml here already all the microservices like accounts, loans and cards, they are already using these microservice-configserver-config. That's why in the same place I'm going to create new environment variables,SPRING\_DATASOURCE\_USERNAME and similarly SPRING\_DATASOURCE\_PASSWORD.

microservice-configserver-config:  
 extends:  
 service: microservice-base-config  
 environment:  
 SPRING\_PROFILES\_ACTIVE: default  
 SPRING\_CONFIG\_IMPORT: configserver:http://configserver:8071/  
 SPRING\_DATASOURCE\_USERNAME: "root"  
 SPRING\_DATASOURCE\_PASSWORD: "root"

So these credentials we set as of now the same values for all microservices.

That's why we don't have to repeat these environment variables again and again inside the

**docker-compose.yaml file**.

The other important environment variable that we need to mention inside our microservices is, we need to mention the data source URL details. So since that database URL is

going to different for different microservices. We need to create a separate environment property variable inside each of the microservices.

So just after the spring application name environment variable, I'm going to create a new environment variable which is, SPRING\_DATASOURCE\_URL to this environment variable,

I'm going to pass the complete endpoint details of my database. You can get this value from the **application.yml** itself. Like you can take this complete value and mention the same inside the docker compose file. But here the communication will not happen with the help of localhost.

Instead, the communication between the containers will happen with the help of service name that we have created.

Here what is the service name for accounts database? It is the same as accounts database.

So let me replace the localhost with **accounts database**. So with this now my accounts microservice should be able to connect with the database accounts database container.

accounts:  
 image: "boss215/accounts:s7"  
 container\_name: accounts-ms  
 ports:  
 - "8080:8080"  
 depends\_on:  
 configserver:  
 condition: service\_healthy  
 environment:  
 SPRING\_APPLICATION\_NAME: "accounts"  
 SPRING\_DATASOURCE\_URL: "jdbc:mysql://accountsdb:3306/accountsdb"  
 extends:  
 file: common-config.yml  
 service: microservice-configserver-config

Same as change for the cards and loans

|  |  |
| --- | --- |
| Cards  environment:  SPRING\_APPLICATION\_NAME: "cards"  SPRING\_DATASOURCE\_URL: "jdbc:mysql://cardsdb:3308/cardsdb" | Loans  environment:  SPRING\_APPLICATION\_NAME: "loans"  SPRING\_DATASOURCE\_URL: "jdbc:mysql://loansdb:3307/loansdb" |

Now the link establishment between the microservice containers and database containers is completed.

The next configuration that we need to do here is, we need to communicate to the Docker compose that my individual microservices are depend on the accounts database or loans database, our cards database respectively.

So for the same just after our environment variables, we can create one more element which is **depends\_on** after depends\_on,

I'm going to go to the next line and mention the service name as accounts database and the condition is service healthy.

depends\_on:  
 cardsdb:  
 condition: service\_healthy

The Docker compose has to make sure the accounts database is started and its health is fine before it is trying to attempt the accounts microservice. So now let me do the same for loans, microservice also. So with this all our docker compose file changes are completed.

**docker-compose.yml file**

services:  
 accontsdb:  
 image: mysql  
 container\_name: accountsdb  
 ports:  
 - "3306:3306"  
 healthcheck:  
 test: ["CMD","mysqladmin", "ping" "-h", "localhost"]  
 interval: 10s  
 timeout: 5s  
 retries: 10  
 start\_period: 10s  
 environment:  
 MYSQL\_ROOT\_PASSWORD: "root"  
 MYSQL\_DATABASE: "accountsdb"  
 MYSQL\_USER: "root"  
 MYSQL\_PASSWORD: "root"  
 extends:  
 file: common-config.yml  
 service: microservice-base-config  
 cardsdb:  
 image: mysql  
 container\_name: cardsdb  
 ports:  
 - "3308:3306"  
 healthcheck:  
 test: [ "CMD","mysqladmin", "ping" "-h", "localhost" ]  
 interval: 10s  
 timeout: 5s  
 retries: 10  
 start\_period: 10s  
 environment:  
 MYSQL\_ROOT\_PASSWORD: "root"  
 MYSQL\_DATABASE: "cardsdb"  
 MYSQL\_USER: "root"  
 MYSQL\_PASSWORD: "root"  
 extends:  
 file: common-config.yml  
 service: microservice-base-config  
  
 loansdb:  
 image: mysql  
 container\_name: loansdb  
 ports:  
 - "3307:3306"  
 healthcheck:  
 test: [ "CMD","mysqladmin", "ping" "-h", "localhost" ]  
 interval: 10s  
 timeout: 5s  
 retries: 10  
 start\_period: 10s  
 environment:  
 MYSQL\_ROOT\_PASSWORD: "root"  
 MYSQL\_DATABASE: "loanssdb"  
 MYSQL\_USER: "root"  
 MYSQL\_PASSWORD: "root"  
 extends:  
 file: common-config.yml  
 service: microservice-base-config  
  
 configserver:  
 image: "boss215/configserver:s6"  
 container\_name: configserver-ms  
 ports:  
 - "8071:8071"  
 depends\_on:  
 rabbit:  
 condition: service\_healthy  
 healthcheck:  
 test: "curl --fail --silent localhost:8071/actuator/health/readiness | grep UP || exit 1"  
 interval: 10s  
 timeout: 5s  
 retries: 10  
 start\_period: 10s  
 extends:  
 file: common-config.yml  
 service: microservice-base-config  
  
 accounts:  
 image: "boss215/accounts:s6"  
 container\_name: accounts-ms  
 ports:  
 - "8080:8080"  
 depends\_on:  
 accountsdb:  
 condition: service\_healthy  
 environment:  
 SPRING\_APPLICATION\_NAME: "accounts"  
 SPRING\_DATASOURCE\_URL: "jdbc:mysql://accountsdb:3306/accountsdb"  
 extends:  
 file: common-config.yml  
 service: microservice-configserver-config  
  
 loans:  
 image: "boss215/loans:s6"  
 container\_name: loans-ms  
 ports:  
 - "8090:8090"  
 depends\_on:  
 loansdb:  
 condition: service\_healthy  
 environment:  
 SPRING\_APPLICATION\_NAME: "loans"  
 SPRING\_DATASOURCE\_URL: "jdbc:mysql://loansdb:3307/loansdb"  
 extends:  
 file: common-config.yml  
 service: microservice-configserver-config  
  
 cards:  
 image: "boss215/cards:s6"  
 container\_name: cards-ms  
 ports:  
 - "9000:9000"  
 depends\_on:  
 cardsdb:  
 condition: service\_healthy  
 environment:  
 SPRING\_APPLICATION\_NAME: "cards"  
 SPRING\_DATASOURCE\_URL: "jdbc:mysql://cardsdb:3308/cardsdb"  
 extends:  
 file: common-config.yml  
 service: microservice-configserver-config  
  
networks:  
 msbank:  
 driver: "bridge"

But if you see here, there is a lot of repetitive information inside my accounts database like **image name, health check and MySQL root password**. So these information is repeated inside my Docker compose file. So that's why I can try to move this to **common-config.yml**. So here what I can do is, I can try to create a new service that extends network deploy service. So let me create a new service and the service name I can keep it as microservice-db-config and this is going to extend network deploy service. And here I'm going to mention all the repetitive information like image. It will be MySQL for all of my database containers. And very similarly I'll try to get the other values. Like port mapping is unique and container name is unique. We can take these health check details along with the MySQL root password. So let me take all these values and mention inside the **common\_config.yml**. So here I'm just mentioning all those details.

microservice-db-config:  
 extends:  
 service: network-deploy-service  
 image: mysql  
  
 healthcheck:  
 test: [ "CMD","mysqladmin", "ping" "-h", "localhost" ]  
 interval: 10s  
 timeout: 5s  
 retries: 10  
 start\_period: 10s  
 environment:  
 MYSQL\_ROOT\_PASSWORD: "root"

and As of now we have mentioned this rabbit service name under the depends on of my config server, but we are no more using the spring cloud bus related changes. But we already deleted this service name. so we will remove from the configserver

And final

**docker-compose.yml**

services:  
 accontsdb:  
 container\_name: accountsdb  
 ports:  
 - "3306:3306"  
 environment:  
 MYSQL\_DATABASE: "accountsdb"  
 extends:  
 file: common-config.yml  
 service: microservice-db-config  
 cardsdb:  
 image: mysql  
 container\_name: cardsdb  
 ports:  
 - "3308:3306"  
  
 environment:  
 MYSQL\_DATABASE: "cardsdb"  
 extends:  
 file: common-config.yml  
 service: microservice-db-config  
  
 loansdb:  
 image: mysql  
 container\_name: loansdb  
 ports:  
 - "3307:3306"  
 environment:  
 MYSQL\_DATABASE: "loanssdb"  
  
 extends:  
 file: common-config.yml  
 service: microservice-db-config  
  
 configserver:  
 image: "boss215/configserver:s6"  
 container\_name: configserver-ms  
 ports:  
 - "8071:8071"  
  
 healthcheck:  
 test: "curl --fail --silent localhost:8071/actuator/health/readiness | grep UP || exit 1"  
 interval: 10s  
 timeout: 5s  
 retries: 10  
 start\_period: 10s  
 extends:  
 file: common-config.yml  
 service: microservice-base-config  
  
 accounts:  
 image: "boss215/accounts:s7"  
 container\_name: accounts-ms  
 ports:  
 - "8080:8080"  
 depends\_on:  
 accountsdb:  
 condition: service\_healthy  
 environment:  
 SPRING\_APPLICATION\_NAME: "accounts"  
 SPRING\_DATASOURCE\_URL: "jdbc:mysql://accountsdb:3306/accountsdb"  
  
 extends:  
 file: common-config.yml  
 service: microservice-configserver-config  
  
 loans:  
 image: "boss215/loans:s7"  
 container\_name: loans-ms  
 ports:  
 - "8090:8090"  
 depends\_on:  
 loansdb:  
 condition: service\_healthy  
 environment:  
 SPRING\_APPLICATION\_NAME: "loans"  
 SPRING\_DATASOURCE\_URL: "jdbc:mysql://loansdb:3307/loansdb"  
 extends:  
 file: common-config.yml  
 service: microservice-configserver-config  
  
 cards:  
 image: "boss215/cards:s7"  
 container\_name: cards-ms  
 ports:  
 - "9000:9000"  
 depends\_on:  
 cardsdb:  
 condition: service\_healthy  
 environment:  
 SPRING\_APPLICATION\_NAME: "cards"  
 SPRING\_DATASOURCE\_URL: "jdbc:mysql://cardsdb:3308/cardsdb"  
 extends:  
 file: common-config.yml  
 service: microservice-configserver-config  
  
networks:  
 msbank:  
 driver: "bridge"

**Running microservices & Mysql DB container using docker compose file**

let's try to validate that Docker compose changes,let me go to the terminal. Inside my terminal I'm going to navigate to the Docker compose folder. So inside this Docker compose we have default folder and inside this folder only we have made all the changes. So the command that we need to run here is docker compose up. So I'm trying to run this command and see whether it is working

**E:\Microservices\section7\docker-compose\default>docker compose up -d**

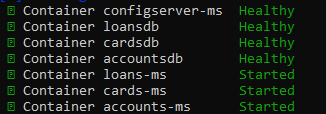
Using this our loans and cards microservices not attached with the db containe bcoz we are trying to expose 3308 and 3307 to outside world but the container they are in their own network they are not trying to connect with our database from external location or from the outside world that’s why we will use here 3306 where our loans,card, accounts db started inside the docker network

SPRING\_DATASOURCE\_URL: "jdbc:mysql://cardsdb:3306/cardsdb"

SPRING\_DATASOURCE\_URL: "jdbc:mysql://loansdb:3306/loansdb"

So now our db will start at same network port of docker which is 3036

**Output**



**Note \* :-** if our database is inside the cloud in such scenario we don’t have to declare loans,cards,accounts database services we can directly define the hostname or public ip which we got from the cloud env.Provide those detail directly at the place of **accountsdb:3306** in the below properties inside the microservices

SPRING\_DATASOURCE\_URL: "jdbc:mysql://accountsdb:3306/accountsdb"

**Service Discovery And Service Registration**

**Brief Inroduction about microservices traffic**

As of now, inside our microservices network, we built three different types of microservices. The very first microservice that we have here is accounts and similar to accounts we also have loans microservice followed by cards microservice.

So as of now you can see these are the three different microservices that are holding my business logic. So these microservices are responsible to store my data, to fetch my data and if some application is trying to send a request to these microservice, they should execute the business logic and send the proper response. So in real world, our microservices applications, they won't get deployed in a different different servers.

It will not be a scenario like suppose there is an external client with the name C1, C2 and C3.If my C1 is dependent on the accounts microservice, it's not like they will be directly calling your accounts. And very similarly, it's not like C2 will be calling your loans microservice followed by cards microservice. So this type of communication is not allowed. So the process that we are going to follow here is, first we will deploy all our microservices into a common microservice network.

**Microservice Network**

**API Gateway**

**External Clients  
c1,c2,c3**

**Loans Microservice**

**Cards Microservice**

**Accounts Microservice**

**External traffic**

**Security  
Audit  
logging**

Like you can see, this is all the microservice network where I will deploy all of my microservices so we can call this network as microservice network or you can call with your own name. So there is a firewall that I will build around my microservices. So for any external clients who want to connect with my microservices, they need to enter using a single entry point.

So if you see, we can assume this can be as a single entry point and they should not be any other entry points on how the external network traffic can enter into our microservices network. The purpose on why we need to maintain a single entry point is, it will allow us to perform any checks related to security and auditing, logging. So all kind of non-functional requirements we can perform at this single entry point that we have right now. So this single entry point is responsible to accept the external traffic. And we usually call this single entry points as gateway. And some people, they will also call it as API gateway. Because to access your APIs, this is the only gateway for the external clients to enter into your microservice network. So whatever requests that we are getting from the external clients like C1, C2, C3, they has to come through this API gateway and we call these traffic as external traffic. So you can see I'm trying to annotate here this traffic we call it as external traffic. So API gateway is going to take care of validating all the external traffic.

So this is one of the component that we need to build inside our microservices.

We're going to discuss about this API gateway in the coming sections. But right now the focus inside this section is, the internal communication sometimes a microservice can be depend on the other microservice.

Suppose think like an external request came to my accounts microservice through this API gateway. Now, in order to send the response to these external application my accounts microservice, it has to connect with the loans microservice and at the same time it also has to connect with the cards microservice. So these communication, we call it as internal communication.So whatever communication is happening inside your microservice network, we call this as internal communication.

So there are two types of traffic like external traffic or external communication or internal traffic or internal communication. Inside this section, we are going to focus on what are the challenges that we may face. While my microservice, they are trying to communicate with each other inside this microservice network. We're not going to talk about external traffic.

Inside this section we will talk about gateway and external traffic in the coming sections. But right now let's focus on internal communication and what are the challenges that we may face and how to overcome that.

**Introduction to the Service Discovery & Registration inside Microservice**

The challenge that we are going to discuss inside this section is, how microservices discover other microservices and how they register themselves into the microservices network. So let's try to understand more details about this challenge.

For the same, I'm going to ask some questions here.

**The very first question** is how a microservice is going to locate other microservices inside a network. like we discussed in the previous lecture, all our microservices will be deployed inside a microservice network. If one instance of a microservice want to connect with other instance of a microservice,how it is going to connect with it.

You may answer here saying that every microservice will have its own endpoint, will have its own port number. Why can't my other microservice can use those details and try to connect with it? I agree with that.

But this approach may work to some extent inside monolithic applications, but inside microservices will be creating and destroying the containers based upon our own requirements. That means the endpoints will keep changing dynamically whenever we try to scale up our microservice instance or whenever we are trying to destroy a particular microservice container because it's not responding.

In such scenarios, how other microservice will know what is the IP address of a specific microservice. So it is going to be impossible to track such details.

So there is a challenge here.

**The next question** that I want to ask here is, how a new service instance can enter into the Microservice network.

For example, while the production deployment we might have started with a one instance of account, one instance of loans and one instance of cards microservice.

But later on we felt that the traffic is very high and we decided to increase the number of instances from 1 to 5 for all the microservices. So when this scale up is happening, runtime inside the production, how they are going to enter into the network and how they will communicate about their details to the other microservices.

So my accounts microservice might be thinking there is only one lone microservice instance inside the Microservice network. But behind the scenes we created four new microservice instances of loans, microservice and cards microservice. How these details will get communicated to accounts microservice. This scenario we may also face whenever we are trying to replace an unhealthy instance with a new microservice container.

Since we are trying to create a new container, definitely there will be a new IP address will be assigned to it. So with all these dynamic conditions or dynamic situations inside the microservices network, it's very challenge how a new service can enter into the microservice network and communicate about its details

to the other microservices.

If they're not able to communicate about themselves to the other microservice, then they will never be invoked.

So there is no use of scaling up or replacing an unhealthy container. In the similar lines, whenever we have multiple instances of a microservice deployed inside a environment, we need to also

think about how a load balancing will happen between the microservice instance.

For example, microservice instance is trying to call loads microservice but loans

microservice has five different parallel running containers available inside the same environment. So how my accounts microservice will know which instance of the loans microservice has to be invoked.So how that load balancing is going to work.

The scenario should not be like accounts microservice always forwarding the traffic to one instance of the loans microservice and making it super busy while other instances they are enjoying their holidays. So it should not be like that. So with all these questions and challenges, we need to understand how to work on this. So let me try to reveal the solution for these challenges. So to overcome all these challenges, we have concepts and patterns or standards with the name

**service discovery**,

**service registration**

**load balancing**

So we need to make sure our microservice network and all the containers inside the network are following are implemented. These standards and concepts like service discovery, service registration and load balancing. I know you might be scratching your head saying that this fellow again bring new jargons like service discovery, service registration, load balancing.

I don't know what are those? So don't worry. I'm going to make them clear for you.

But for now, I'm assuming you are clear the challenges that we are going to discuss inside this section. So, take your time and try to read these questions and the challenges that we have.

**Why Not Traditional load balancers for Microservices**

For a few seconds

let's think like we don't have concepts like service discovery and service registration, and we are forced to use that traditional approaches inside microservices environment. So with that assumption, let's try to see what are the challenges or disadvantages that we may face if we follow that traditional monolithic approaches inside microservices, internal communication. Usually inside a web network, whenever a service or an application want to communicate with another

service or application, it must be given the necessary information to locate it, such as IP address or instead of IP address, we can also give that DNS name or the domain name. With this information now

let's try to see the scenario of two microservices like accounts and loans are trying to communicate with each other. For simplicity reason

let's assume there is only one instance of loans and one instance of accounts inside my microservices network. Like you can see first I have my accounts microservice installed or deployed inside a server and we call this as upstream service. Because this is the service which is dependent on the another service.

That's why we call this as upstream service.

And the other side, we also have another microservice which is loans microservice, and we call thisservice as downstream service. Because this loans microservice is acting as a dependent service for the accounts microservice. Like you can see here, the loans microservice is deployed at an IP number.127.54.37.23.

Now if my accounts microservice want to communicate with the loans microservice, there are two options.

One is inside my accounts microservice I can hardcode the IP details where my loans microservice is deployed.

Or we can also mention that domain details are DNS name inside the accounts microservice code. So with that, the internal communication between microservices will use either the hostname DNS or IP address. So there is no service discovery at the load balancing involved here.In this kind of situations we call loans. microservice will be acting as a backing service with respect to the accounts microservice. Because without the loans microservice my accounts microservice cannot send a successful response to the clients who is trying to get the consolidated information of accounts and loans microservice.

So that's why loans microservice will act as a backing service to the accounts

microservice. So mentioning the IP address or DNS name at the hostname will work perfectly. When we have only one instance of loans microservice running inside our microservice network. And even managing the DNS name and its corresponding IP address mapping is going to be straightforward. If you are not clear about what is DNS. So DNS is like using a domain name instead of IP address. So behind the scenes, your platform team or operations team members, they will do a mapping of your DNS name to the correspondent IP address. So this way in future, if my loans microservice IP address changes, I'm going to change the mapping

between the DNS and IP address and there won't be any code changes on the accounts microservice if they are using that DNS name.

Whereas if they hardcode the IP address, whenever the loans microservice IP address is being changed, the same needs to be changed on the accounts microservice. So to avoid these kind of situations, usually inside the web applications, communications, we rely on that DNS names. So here there is a single IP address.

Someone will map this IP address to a DNS name and using the same DNS name, my accounts microservice can connect with the loans microservice, but in a cloud environment where we are going to deploy multiple instances of a service with each instance having its own unique IP address, this is not going to work. Because someone has to maintain the mapping between the DNS and the list of IP addresses where we have deployed. To some extent, we can overcome this challenge like we can have someone updating these DNS records with the multiple IP address and we can also rely on the algorithms like round Robin to load balance the traffic between multiple instances of loans microservice.So updating that DNS records with the multiple IP addresses will work perfectly for monolithic applications or soa applications because we will have very limited number of services in traditional applications. But inside the microservices network, this approach may not be suitable because we are going to use a containers and these containers deployment will keep changing frequently. So this rapid change make it difficult to maintain the accurate DNS records and ensure efficient communication between the microservices.

So like I said before, whenever we are getting more traffic, we are going to onboard the more instances of loans microservice. And the same when the traffic is down, we are going to delete the few of the running instances of loans microservice.

And very similarly, if one of the loans microservice instance is not responding, we are going to delete it and replace it with a new instance.

So when you are doing all these rapid changes inside the cloud environment, it is an impossible task to maintain the accurate mapping between your DNS and the multiple IP addresses of your loans microservice instances. Because unlike monolithic applications where we are going to deploy the applications inside a physical

machine or a long running virtual machines, in such scenarios, the IP address will be static and it will never change until unless you are trying to change it.

Whereas with cloud native applications and microservices, the instances are the containers will have shorter life spans like we discussed during the auto scaling scenario or whenever the container is not responding.

In all such scenarios, we're going to bring new containers and new instances which will have new IP addresses. With all these problems, the primary challenge that we are going to face here is, the DNS records will not be up to the date with the latest IP address details of the banking services like loans microservice Inside the next slide, I tried to explain this visually. For example, first we are going to have certain client applications, our client services who are trying to invoke other microservices with the help of DNS name. Like you can see here, someone is trying to invoke the accounts microservice.

Similarly, someone is trying to invoke the cards microservice and loans microservice with a different different DNS names. Whenever the request received to this DNS behind the scenes, there will be a traditional load balancer. Try to monitor the request and based upon the incoming requests along with the DNS names, it will try

to look for the what are the exact details.

So you can see there is a primary load balancer here which is responsible to process all the requests coming to the DNS with the name services.easybank.com.

So based upon the path that it sees inside the request, it will try to look for the corresponding physical IP address by looking into the routing tables.

So someone might have configured all the details inside the routing tables.

Once my load balancer identifies what are the exact details, the same request will be forwarded to the actual services, so there will be also secondary load balancer which will keep checking the health status of the primary load balancer.

For some reason, if my primary load balancer is not responding, then immediately my secondary load balancer will act as a primary load balancer.

This way we are trying to overcome the scenarios where the primary load balancer is down, so there is always a backup load balancer. But there are certain problems with these traditional load balancer because we are using in between traditional load balancer.

Let's try to understand what are those disadvantages. Usually inside that traditional applications, each instance of a microservice are service will be deployed

in one or more application servers. The number of these application servers usually will be static, and even in the case of restoration,

it would be restored to the same state with the same IP and other configurations.

This type of setup will work perfectly for monolithic applications and soa based applications, where they have a relatively small number of services running with a group of static servers and static IP

address. But like I said, this is not going to work for the cloud based microservices applications, because the IP addresses are going to change.

That means your traditional load balancer will not be able to manage the traffic if the IP addresses are keep changing dynamically.

So with this limitation inside the traditional load balancers, there are many disadvantages. The very first one is limited horizontal scalability and license cost.

Whenever we want to use the traditional load balancers, we should definitely know the, what are the IP address well ahead. So the same we need to configure inside the routing tables and using the same routing tables, my traditional

load balancer can forward the traffic to the corresponding services. Since we need to maintain those IP address details manually inside the routing table, we cannot really perform a lot of scalability like dynamic scale up and scale down. We cannot perform inside our microservices. So that will be a huge limitation if you follow the traditional load balancer and at the same time these traditional load balancer, they are not free of cost. Every cloud provider, they provide these traditional load balancer with few licensing costs. So they will also request some budget to set up inside your microservices network. And the next disadvantage that we have here is single point of failure. We saw in the previous slide there are two load balancers.

If my primary load balancer fail, then my backup load balancer like secondary load balancer will come into picture. What if both of them fail whenever we use this traditional load balancers, we can't scale them easily like inside a cluster environment.

So with these reasons, it is going to be a single point of failure. And with that, all your incoming requests will be choked at a centralized location.

And the next major disadvantage that we have is someone has to manually manage the updating the IPs configurations inside the traditional load balancer, which is an impossible scenario inside microservices applications.

On top of these advantages, these traditional load balancers are complex in nature.

Someone has to maintain them manually and they are not container friendly.

Whenever we are using Docker containers, we can't use traditional load balancers with the primary reason that Docker containers can be created or destroyed at any point of time. So the summary from this discussion is, whenever you are using traditional load balancer, the biggest challenge will be that someone has to manually maintain the routing tables, which is an impossible task inside the microservices network. Because containers and microservices are ephemeral in nature. So what is ephemeral? Ephemeral means they are short lived and they will be disposed at any point of time. I hope you are clear with the discussion about the traditional load balancers. Inside the next lecture, let's try to understand how to overcome this challenge for cloud native applications and microservices

applications.

**Service Discovery & Registration Inside microservice**

For cloud native applications , **service discovery** is the perfect solution. It involves tracking and storing information about all running service instances in a **service registry**

Whenever a new instance is created, it should be registered in the registry, and when it is terminated, it should be appropriately removed automatically.

The registry acknowledges that multiple instances of the same applications can be active simultaneously. When an application needs to communicate with a backing service , it performs a lookup in the registry to determine the IP address to connect to . If multiple instances are available a load balancing strategy is employed to evenly distribute the workload among them.

So this load balancing and service discovery can happen in two different styles.

One is client side service discovery, and the other one is server side service discovery.

So these are the two different approaches.

So let's try to understand more details here. Inside Microservices network are cloud native applications. We know that the IP addresses will be dynamically changing because we are going to change the number of instances whenever we want to perform auto scaling or in the scenarios of failures.

So in these kind of dynamic situations, microservices patterns like service discovery and registration is a way for applications and microservices to locate each other on a network.

So let's try to understand what are the components involved inside these patterns. The very first component is there will be a

**centralized server or a multiple servers** that maintains

a global view of address, just like how we have config server which is responsible to maintain all the configurations at a centralized location. Very similarly, we are going to create a central server that maintains the details of all the running

services. So how it is going to maintain is whenever a particular micro service instance is trying to get started very first time, it is a responsibility of the microservice itself to connect to the central server and to register their address when they start and ready.

So the microservice instance that started it has to communicate its IP address port details to the

central server and with that the central server will maintain that information inside a service registry. And once the microservice is started, my central server also expects a regular heartbeats from my

individual microservices confirming about their health. If my central server, which is acting as a service registry and service discovery, if it is not receiving the regular heartbeats, then it assumes the health of a particular microservice instance is not good and it is going to delete its IP details and all the address details from the service registry. And very similarly, whenever the microservices are being shut down or being removed. They should also re-register their address details from the central server.

So since my central server now has a global view of address, any other microservice which want to communicate with the backing microservice, it can connect with the central server and ask what are the IP address of so and so microservice. So the summary here is the service discovery and service registrations deals with the problems about how microservices talk with each other.

So how this pattern is going to be implemented is, we are going to create a separate central server which is responsible to implement this service discovery and service registration.

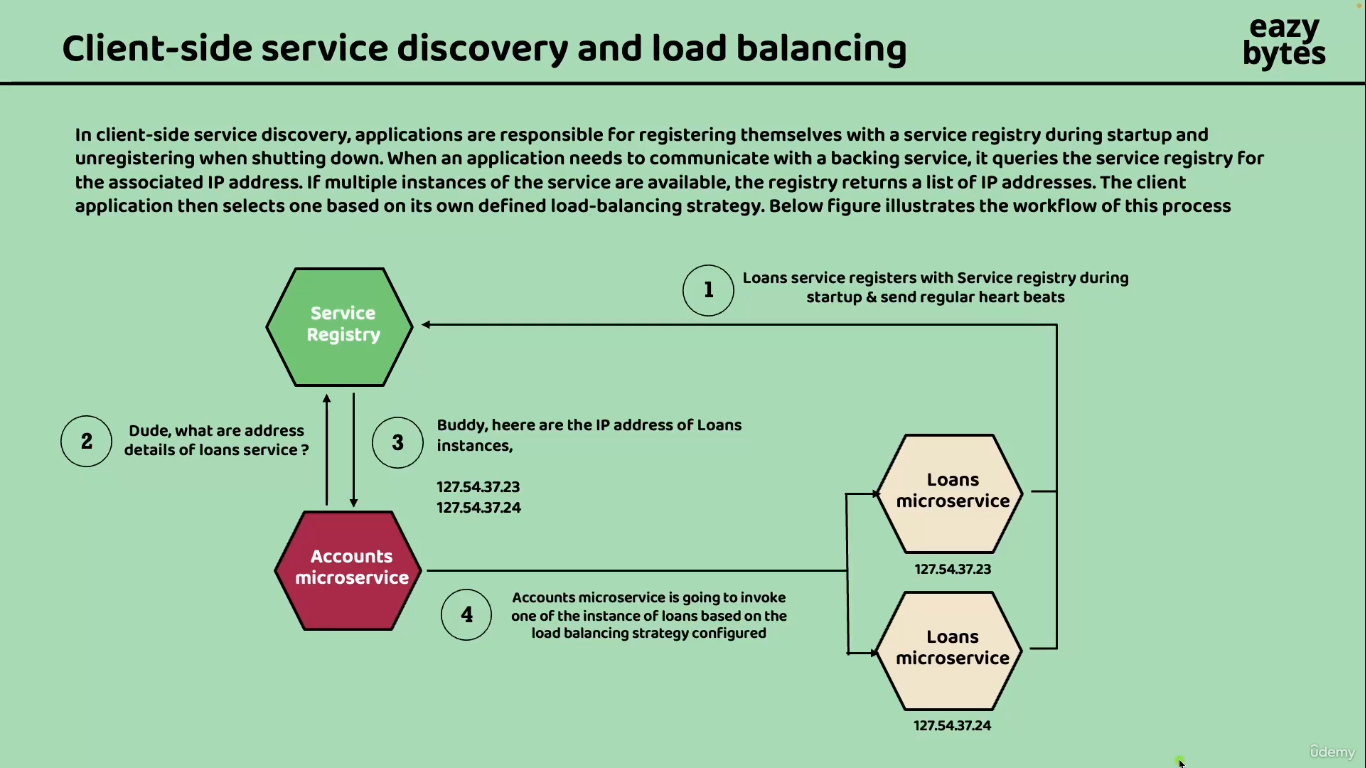
Previously I said that the service discovery can happen at the client side and the server side.

Inside this section we are going to focus on client side service discovery.

**Service discovery & registrations deals with the problems about how microservices talk to each other,**

**i.e. perform API calls**

**How Client Side Service Discovery & Load Balancing works**

****

And once the service registry is updated, when an application or a microservice want to communicate

with a backing service like an accounts microservice want to communicate with the loans microservice.

In such scenarios, the account microservice will queries the service registry for the associated IP

addresses of loans microservice. If multiple instances of the loan service are available inside the registry, the registry is going to return the list of IP addresses to the accounts

microservice. Now the responsibility of selecting one of the IP address of the backing service or IP address of the loan service is on to the client microservice, which is accounts microservice.

Since the responsibility is on the client side to decide to which instance of the backing service it

needs to redirect the request that's why we call this approach as client side service discovery.

Let me try to show how this flow works with a detailed explanation. In the very first step, we will have the service registry. So before we try to start all our microservices, first we need to make sure the service registry is started. So this is a very separate server running inside your microservice network.

Now inside my microservice network, I'm trying to start two instances of loans microservice. And each

of them has two different IP addresses. during the startup of these respective instances of loans microservice, these instances will connect with the service registry and update their IP addresses, hostname or port number. All those details will be registered with the service registry.

And apart from the startup, they should also send the regular heartbeat signals to the service registry

confirming their health status. So once we have these details of loans, microservice inside the service registry. Now I have my client, microservice with the name accounts microservice because this microservice want to connect with the loans. Microservice So now my accounts, microservice has a problem here. It don't know what are the details of loans microservice and how many instances of loans microservice are running. That's why it is simply going to ask the service registry, Hey dude, what are the address details of loans microservice. I want to know those details now. The service registry

after validating the accounts microservice request, it is going to return the Hey buddy, these are the IP addresses of loans instances, so it will give the list of IP addresses.

So now my account microservice has a problem here, so it receives multiple IP addresses of loans

instance. In real prod applications, we may receive more than ten IPS because more than ten instances

of loans microservice might be running. In this kind of situations.

My account microservice is going to follow a load balancing strategy and based upon this load balancing

strategy, it is going to forward the request to one of the loans microservice instance.

So if you see here the responsibility of load balancing and service discovery is on the client microservice

itself, which is account microservice. So the account microservice has to look for the service details.

So we call that as service discovery, which is happening in the step two and step three.

And once the service discovery is completed, the load balancing also is going to be taken care by the

client microservice itself and post that the invocation to the actual microservice will happen.

Since this responsibility is on the client side, we call this pattern as client side service discovery

and load balancing

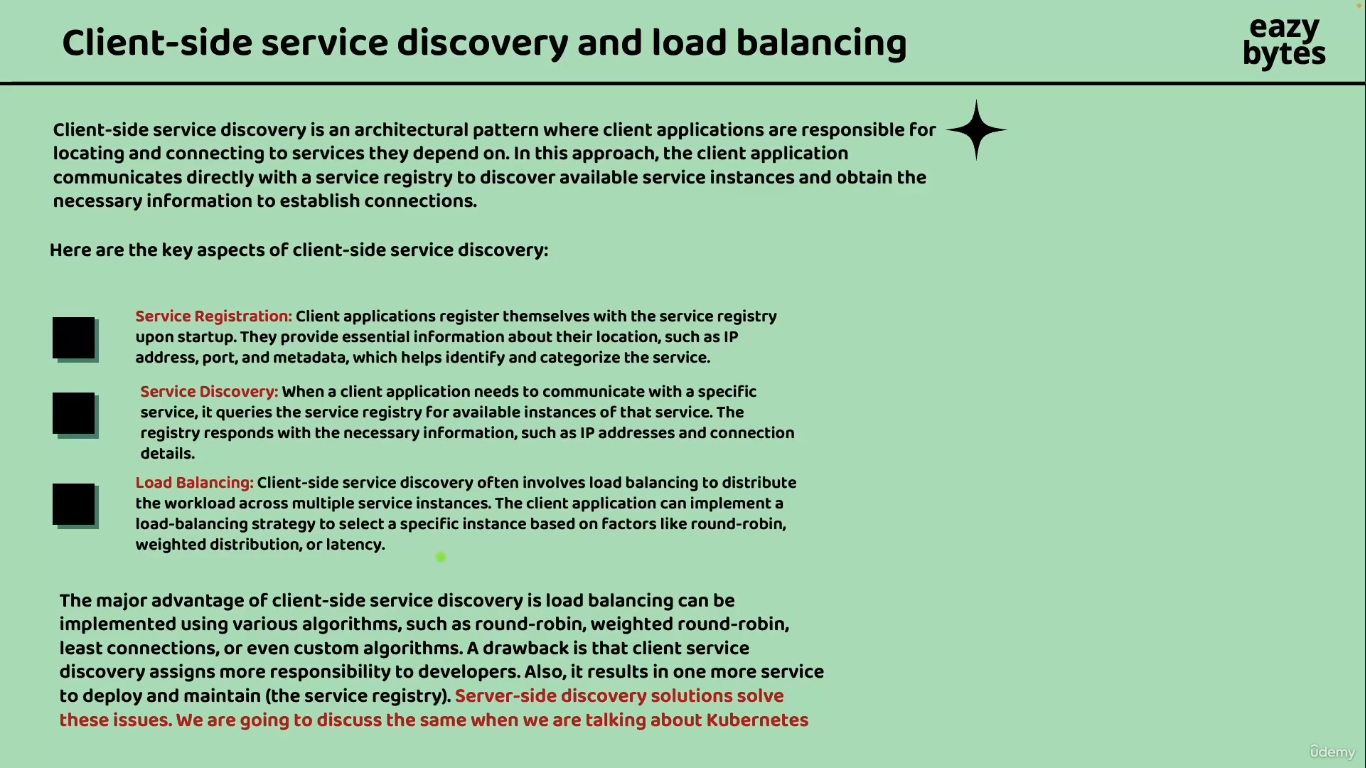
I hope this is clear. The same information I tried to mention inside the next slide.

What is a client side service discovery, what will happen during the service registration,

what will happen during the service Discovery and what will happen during the load balancing.

So the main advantage with these client side service discovery is, you have an option to follow multiple

Strategies around the load balancing. You can follow the algorithm of round robin, which will take care of forwarding the request on a round robin fashion. And very similarly, we can also follow the weighted round robin or if needed, we can also follow the strategy like least connections, like whichever instance has least connections to the same, I want to forward the request. So that is called least connection strategy. Or you can write your own custom algorithms. So that's a primary advantage that we have with the client side service discovery. Whereas the drawback with these client side service discovery is, it assigns more responsibility to the developers. The reason is inside our individual microservices, we need to make some code changes so that they can talk with the service registration, they can perform some load balancing. So since there is a responsibility on the developers to achieve these client side service discovery, it is considered as its drawback



And at the same time, whenever we want to use client side service discovery, we need to maintain a

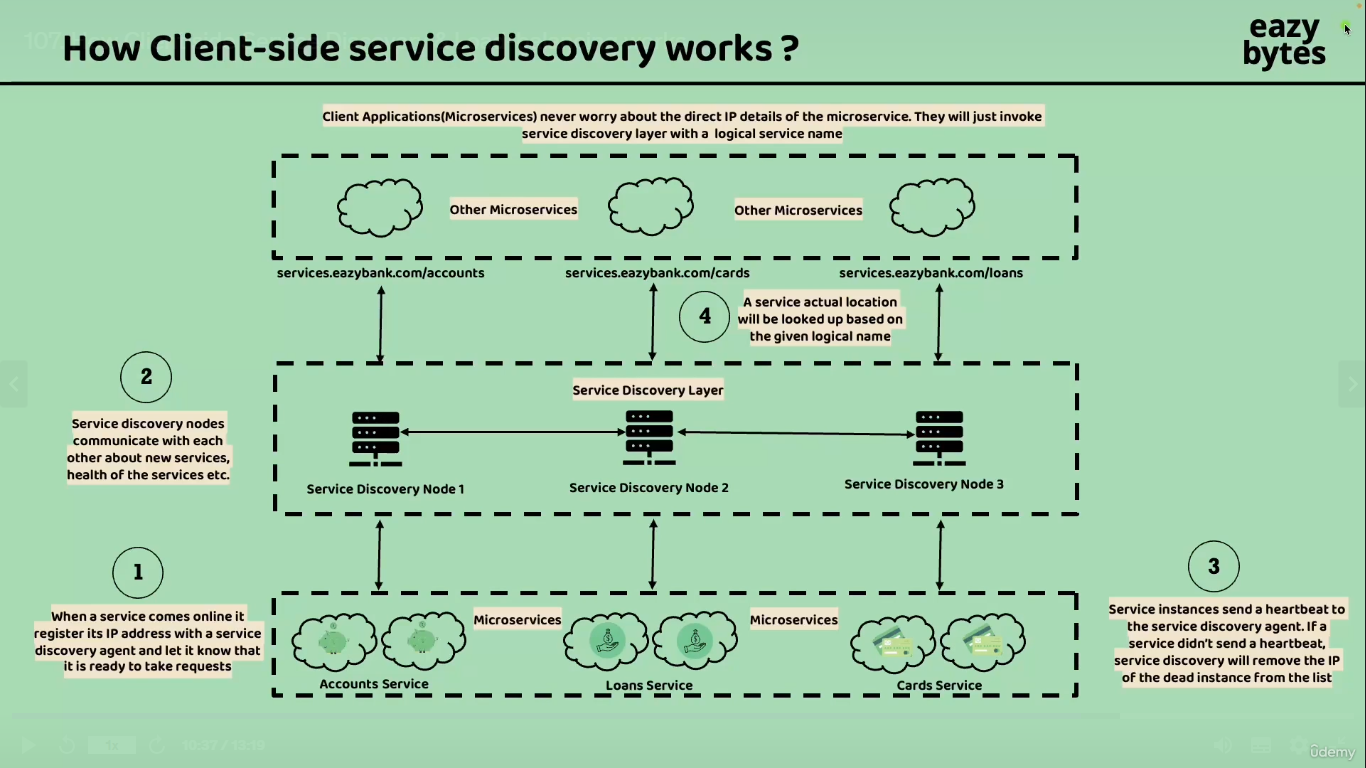
centralized server which is going to maintain the service registration and support the service discovery.

To overcome these challenges of client side service discovery.

We also have server side discovery solution, which we can implement whenever we are trying to deploy

our microservices inside a Kubernetes environment. Again, maintaining your microservices inside Kubernetes, deploying them into Kubernetes cluster. It involves a lot of team effort and you may need separate operations team and platform team who has a knowledge around the Kubernetes. And even mentioning the Kubernetes cluster inside the cloud is not going to be free. It is going to require a lot of budget on your project side. So if you ask me who will use client side and who will use server side, I would say the projects who can afford Kubernetes cluster and maintaining them inside the cloud environments. They can use server side discovery, whereas if projects they cannot afford the costly maintenance of Kubernetes clusters, they can go with the client side service discovery where they can develop their own centralized server that is responsible for service, registration and service discovery.

But here it may sounds like very complex for you developing these separate server and implementing these service registration, service discovery and load balancing with the help of client side service discovery approach. But don't worry, it's not going to be super complex. It is going to be super, super simple with the help of **Spring Cloud**. So inside Spring Cloud, we have various options for incorporating these client side service discovery inside our spring boot based microservices.



So more details I'm going to discuss in the coming lectures Before I try to close this lecture, let

me show you the different perspective how these client side service discovery works. So that you will

have more details about the client side service discovery.

So whenever we are following the client side service, Discovery first will be your individual microservice

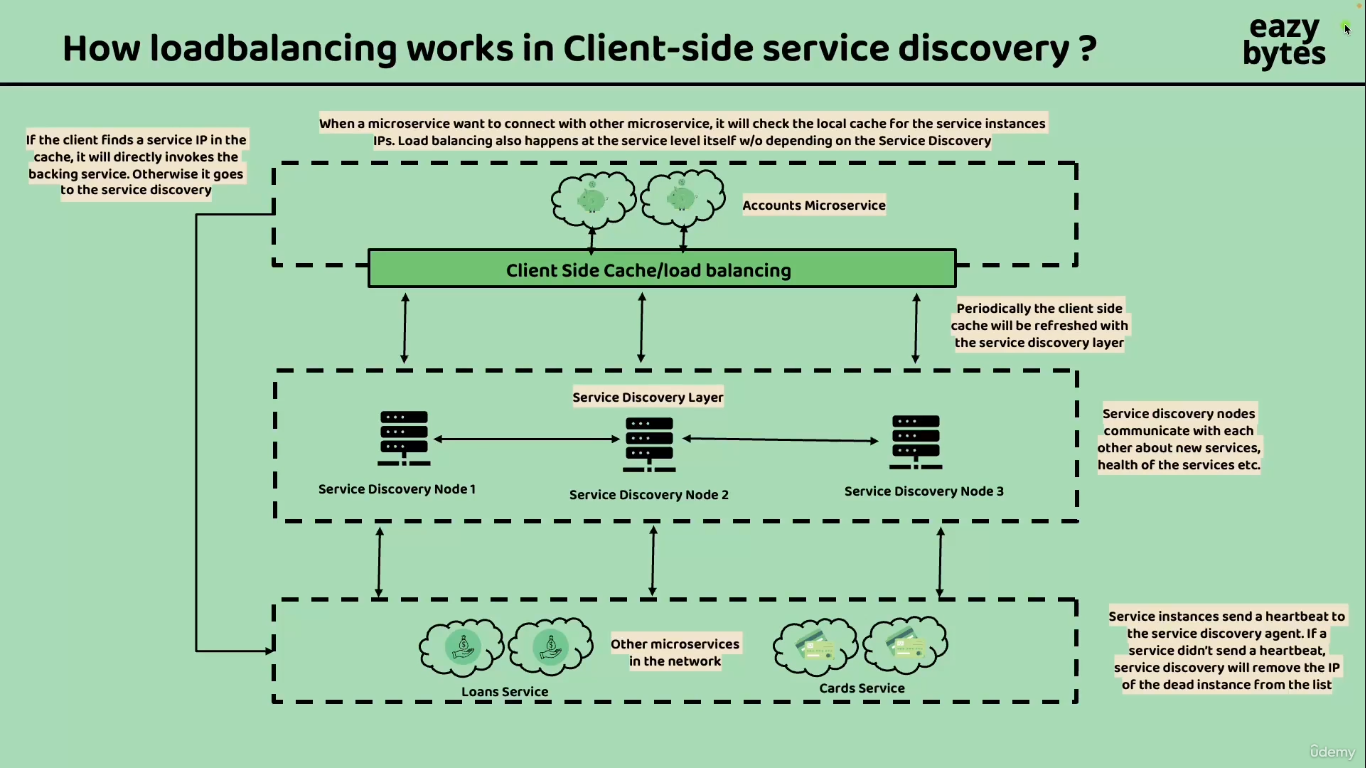
where they are trying to get started inside your microservices network. Like you can see there are two instances of accounts microservice, there are two instances of loans and cards microservice And just like how we have these microservices as we'll also have a supporting component or a centralized server which is responsible to maintain the service registry and service discovery. So these centralized server, we call it as service discovery layer. Inside this layer we can deploy any number of instances of service discovery node based upon the load

that you can expect. We can create any number of instances and deploy them inside our microservices network. And one of the beauty of this service discovery layer is no one has to maintain the IP addresses of

the microservices manually, like we discussed inside the traditional load balancer. So that manual task we don't need to follow here. Instead, my individual microservices during the startup they will communicate with the service discovery layer and try to register themselves like, this is my IP address, this is my hostname, this is my port number. And the beauty here is, even if your individual microservice is trying to register the details with one of the service discovery, the same details will be shared with the other service discovery nodes immediately based upon the gossip protocol. So based upon this gossip protocol, all your service discovery nodes inside your service discovery layer, they are going to communicate regularly and they will keep themselves updated with all the details that are being received from the individual microservices.

For example, whenever my account microservice is trying to register its details with the service node one, the same details will be shared to the other service nodes also immediately. And this is how all the nodes will have the same and updated information at any point of time. And apart from the startup, my individual microservices, they are also going to send the heartbeat signals regularly to the service discovery layer.

So once this setup is completed. Now my other microservices like who want to communicate with my accounts microservice, loans microservice or cards microservice. They will send a request to the service discovery layer by using a logical path. Like you can see here, whatever microservice who want to communicate with the accounts microservice, they are going to send a logical path with a DNS name along with the Path accounts. So based upon this logical name, my service discovery layer is going to return the list of IPS to the client microservice and based upon those IP address, the load balancing will happen at the client side and they will try to communicate with the actual microservice. So this way my client applications are microservice. They never have to worry about the direct IP details of the microservice. They just invoke the service discovery layer with the logical service name and post that based upon the actual IP address, they are going to invoke the actual microservices. So now here you may have question like where is the client side load balancing is going to happen. As of now, you can see inside below slide only the service discovery is right now happening.



So let's try to see how the client side load balancing will happen.

If you see inside this slide, there is a scenario where my accounts microservice is trying to connect

with the other microservices like loans and cards microservice. So all the story is same.

Like all the individual microservices, they will try to register with the service discovery layer and my accounts microservice whenever it want to connect with the loans microservice, first, it will check whether it has any details of the loans microservice in its own cache. If there are no cache details available regarding the details of the loans microservice or cards microservice, it is going to send a request to the service discovery layer which will return the list of IP addresses. Now, at this point of time, my accounts microservice is going to cache them so that for each and every request it don't have to rely on the service discovery layer.

This way we are also going to reduce the burden on the service discovery layer. Think of a scenario where you have 100 different microservices and if they try to communicate with the service discovery layer for each and every request, it is going to be a lot of burden on your service discovery layer. That's why all my client microservices applications, they are going to invoke the service discovery layer very first time once they have the complete details of all the instances of loans microservice or cards microservice, they are going to cast them inside their local memory. And for all the future requests, they are going to use the IP addresses present inside the cache. But here you may have question like what if the IP addresses change?

That's why to overcome this challenge behind the scenes, this client side service discovery makes sure the cache is being refreshed for every few seconds like 10s or 15 seconds or 20s. On top of that, whenever an exception happen. If my accounts microservice is trying to communicate with other microservices with the help of client side caching, then immediately it is going to invalidate the client cache and it will try to get the

latest IP address details from the service discovery layer. So all this is going to happen behind the scenes automatically.We don't have to code these as developers.

We just have to know what projects we have to use inside Spring Cloud and how to configure them inside our microservices network. I hope this is clear on how these client side service discovery is going to work.

**Spring Cloud Support for Client-Side service discovery & Registration**

The next immediate question that you may have here is, how I can implement this client side service discovery

inside my microservices network. With the help of **Spring Cloud project**, implementing service discovery and registration inside a microservice network is going to be super, super simple. We are going to implement the same in the coming lectures and in the same process you will realize how easy it is. So there are multiple components we need to use from the spring cloud project.

So the very first component that we need to use is **Spring Cloud Netflix's Eureka**! So this is a service which will act as a central server, which is responsible for service registration and service discovery. In simple words, we can call this as a service discovery agent. Now for the load balancing, we are going to use **Spring Cloud Load Balancer** library and using the same library, we are going to perform the client side load balancing. And at last we are also going to use

**Netflix Feign client** Once we identify actual service details with the help of Eureka Agent and Load Balancer, we need some library to communicate with the other service, just like how we have REST template and web client inside the spring core framework. Very similarly, we can use this library which is

**Netflix Feign client** :- to perform the actual invocation of REST APIs whenever we are trying to have a communication between the microservices. So these components are the mostly used components by the industry. That's why we are going to use the same inside the course. But apart from these components, we also have other projects inside the industry.

For example, instead of **Eureka**, we can also use other products like **Etcd**, **Consul** and **Apache Zookeeper**.

So these products are also good. But since we are right now building the microservices with the help of Spring Boot, it will be a wise decision to choose one of the project within the same ecosystem, so that the integration with the other projects inside the same ecosystem is going to be super, super easy. And right now we decided to use spring **cloud load balancer** for client side load balancing. In some old projects or in some projects where they are using the older versions of Spring Boot. You may see they are using **Netflix Ribbon** for client side load balancing. This is also good and stable product, but we're not going to use that because, the Netflix Ribbon library right now, it is in the maintenance mode and unfortunately there are no new enhancements or upgrades are happening. The reason is we have a latest library, which is **Spring Cloud Load Balancer,** which came as a replacement for the **Netflix Ribbon client**.

So that's why we are going to use Spring Cloud load Balancer, anytime if you listen about Netflix Ribbon or any other products, don't worry on why we have not used them inside our microservices network because we are trying to use the most stable and most recent products.

So on a high level, whenever we try to implement these client side service discovery inside our microservices applications, we are going to get a lot of benefits, like there are no limitations on availability. We can have multiple nodes of service discovery nodes we can deploy inside the service discovery layer and post that the peer to peer communication we can achieve with the help of service discovery agent. And next, the most important advantage, which is all the IPs configurations and load balancers, they can change dynamically, but still the microservices communication is going to happen because all these configurations and IPs are going to be updated dynamically behind the scenes by my service discovery and service registration components.

And apart from these advantages, whenever we follow these client side service discovery, the communication

between the microservices is going to be **fault tolerant** and **resilient** in nature. We are going to discuss in the coming lectures how we can make the communication between the microservice as fault tolerant and resilient.

**Extra Info About Eureka**

So let me quickly show you about these projects inside the official spring website. So here I came to the spring.io project. If you can run this projects here we have Spring Cloud project and this spring cloud project we have many other sub projects. The projects right now we are interested is, Spring Cloud Netflix. If you can click on this Spring Cloud Netflix, it will open in a new tab. And very similarly, we are also interested on Spring Cloud, Open Feign. So like I said, Spring Cloud Open Feign we can use to communicate with the other REST APIs just like rest template and web client. So more details to follow around this Open Feign.

And very similarly, we also have Spring Cloud Netflix. So inside the Spring Cloud Netflix, there are many components like Eureka Server which will act as a service discovery agent and at the same time it is also going to provide the load balancer with the

help of spring cloud load balancer. So if we can scroll down, you can see Spring Cloud Netflix features like it is going to support the service discovery with the help of Eureka Server. So here you may have a question like why there is a name Netflix, we all know what is Netflix? So it's a ott platform where we see our favorite movies and web series. So there is a story on why we have this Netflix name here. To tell this story

let me go to the blog, which is written by the official tech team from the Netflix. So inside this blog you can see all the history is present like back in the 2007 when Netflix is trying to build a lot of services that can run on the cloud infrastructure. They build a lot of libraries internally, like Ribbon for Load balancing, Eureka for service discovery and Hystrix for the fault tolerance and to stitch all these components they also used another library with the name generator. But around 2012 they open sourced all these components by donating them to the spring community. So with all the libraries donated by the Netflix team to the Spring community, a new project came into picture in 2015 with the name Spring Cloud Netflix. So this project developed by the spring team itself, based upon the libraries donated by the Netflix since the spring team. They want to acknowledge the contributions by the Netflix team. They have mentioned these Netflix inside the project name. After 2015,

there are many updates came to the spring cloud Netflix. As of now you can see we have the version which is 4.0.2. So from 2015 to the present day, there are many updates happen inside the Spring Cloud Netflix.

Since these updates are happening based upon the latest trends and in the right direction.

Netflix team also, they started using Spring Boot as their core Java framework from 2018. And along with the Spring boot framework, they also leverage Spring Cloud Netflix to a great extent from 2018. So if you can scroll down here, initially the setup inside the Netflix used to be like you can see on the left hand side. So there used to be a governator which can monitor all the components. So for client side load balancing, there used to use ribbon and service discovery, Eureka and fault tolerance they used to use **hystrix**.

But now with the latest trends happening, there are many advancements happened on the spring cloud

project. That's why the Netflix team, they are using Spring boot and along with the spring cloud load balancer

instead of Ribbon, followed by Spring Cloud Eureka. And for fault tolerance. They are not using Hystrix because it is outdated. Now we have Resilience4j to implement fault tolerance. So if you see we are going to use the same tech stack inside our course also, we are going to use spring cloud load balancer.

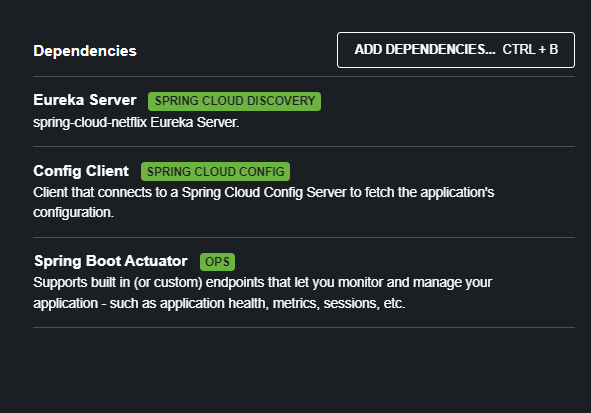
We are also going to use Resiliency4j. And on top of that, we are also going to use Spring Cloud, Open feign to establish the communications between the microservices. So I'm going to mention these URL inside the GitHub repo and at the same time I'm also going to attach this URL details to this lecture. So please try to refer the same and try to read this story in detail and this gives you some confidence that you're not learning something very old. You are learning mostly used, mostly stable, and most latest components inside this course. If Netflix itself is using, then any project can use because the traffic handled by the Netflix is

very huge. So I'm assuming you are clear on what components we are going to use to build the client side service discovery inside our microservices.

**Setup Service Discovery Agent Using Eureka Server**



we are trying to build a service discovery agent that's why we need to select these Eureka server dependency. After adding this dependency, I'm also going to add config client dependency because my Eureka server will connect to the config server to load all the properties related to the Eureka server.

So that's why I'm going to add this config client dependency. And after this I'm going to add the actuator dependency.With these three dependencies, we should be good.  


**EurekaServer.yml**

---------------  
server:  
 port: 8070  
  
eureka:  
 instance:  
 hostname: localhost  
 client:  
 fetchRegistry: false  
 registerWithEureka: false  
 serviceUrl:  
 defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/

The very first one it is pretty clear we want our Eureka server to get started at the port 8070 post

that I have started a new element which is Eureka and a Eureka.

I have mentioned instance. After instant I have mentioned a element with the name hostname and here I'm mentioning the localhost. So with this eureka.instance.hostname we are telling to the Eureka server what host name it has to consider. Obviously inside our local system, the host name is going to be localhost. And after this localhost configuration, I have mentioned

**eureka.client.fetch** registry as false

and register with Eureka as false. So let's try to understand what is the purpose of these properties.

Fetch registry I'm mentioning as false. So by default it is true here I'm trying to set false. The reason is I don't want my Eureka server to fetch the registry details of other microservices. My Eureka server never going to call the microservices. That's why it is unnecessary to fetch all the registry details of the microservices. That's why I'm mentioning these value as false. Whereas in our individual microservices like accounts, loans and cards, this value should be true and which is the default behavior. Since I want to change the default behavior here, I am setting this value as false. Now register with Eureka. So I'm telling to my Eureka server, Don't register with yourself and don't expose your details into the registry of the service Discovery. Since we don't want our Eureka server to register with itself, we have to mention these property as false. And at last I have mentioned eureka.client.service URL and this service URL,

I have mentioned one more element which is default zone. And here I have mentioned a property which is the URL where my Eureka server is going to expose its functionality and other microservices they can try to connect to register their details or to discover other service details. So you can see here the host name I'm trying to fetch from these properties that's why I have mentioned the variable name as eureka.instance.hostname and the port number from this variable server.port. And at last there is a path which is /eureka. So these are default service URL that we want to consider. So with these properties now during the startup, my Eureka server is going to connect with the config server and fetch all these properties and start with these eureka related configurations

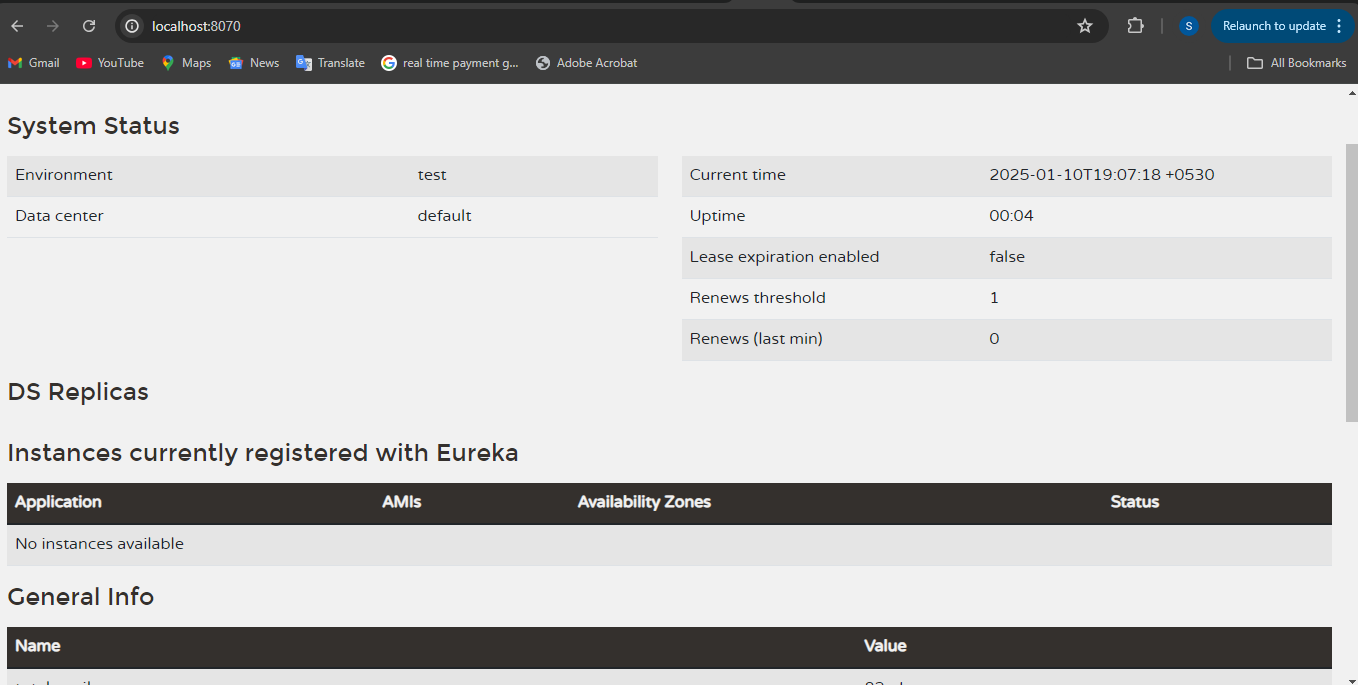
Now to validate if my Eureka server started perfectly or not, I'm going to access a path which is **localhost:8070.** So this is the route path we need to access the same. So as soon as I access, you can see I'm getting the Eureka dashboard as of now, this is the dashboard provided by the Spring Eureka Server itself. I have not made any code changes to build this page, but still I'm able to access it.

The purpose of this Eureka dashboard is once the other microservices, they started registering their

details with the Eureka Server. We can see all those details inside this dashboard.

So as of now it is empty. Like there are no instances currently registered with Eureka because as of now we have not established the link between other microservices to Eureka Server. But this is a good start.

This confirms that our Eureka Server setup is successful. I hope you followed all the steps for your reference, I also mentioned all the steps that we have followed inside this slide. Please refer to the same whenever you want to quickly refresh all the steps that we have followed. With this, we successfully set up the Eureka Server. The next step, obviously is, to establish a link between our individual microservices with the service Discovery agent, which is Eureka Server.



**Make Code Changes in Accounts Microservices to connect Eureka Server**

**Step 1** :- Add Eureka Dependency in accounts microservice

<dependency>

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

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

</dependency>

Step 2 :- Add some properties inside the account property file

eureka:  
 instance:  
 preferIpAddress: true  
 client:  
 fetchRegistry: false  
 registerWithEureka: false  
 serviceUrl:  
 defaultZone: http://localhost:8070/eureka/

we need to add few set of properties inside the application.yml of accounts microservice.

So let me open the **application.yml** file here inside my applications.yml file. First

**preferIpAddress: true**:- true.

Just to make sure our accounts microservice will try to fetch the registry details whenever it is trying

to connect with other microservices. And very similarly, since we want our account microservice to be registered with the Eureka server, I'm trying to mention the value true.

**preferIpAddress: true**

* When this is set to true, the application will register its **IP address** (instead of its hostname) with the Eureka server.
* This is useful in environments where hostname resolution is unreliable or you want clients to connect directly to an IP address.

**fetchRegistry: false**

* When set to false, the application will **not retrieve the list of registered services** from the Eureka server.
* This is typically used for standalone services or services that don't need to discover other services.

**registerWithEureka: false**

* When set to false, the application **will not register itself** as a service with the Eureka server.
* This is useful for client applications that only consume services and don’t need to be discovered by others.

**serviceUrl.defaultZone: http://localhost:8070/eureka/**

* This specifies the URL of the Eureka server that the application communicates with.
* In this case, the Eureka server is running locally on port 8070.
* **defaultZone** is the default key used to define the Eureka service URL.

**How This Configuration Behaves**

1. **No Registration**:
   * The application will not register itself with the Eureka server (registerWithEureka: false).
2. **No Discovery**:
   * The application will not fetch the service registry from the Eureka server (fetchRegistry: false).
3. **Direct Communication**:
   * The application can still manually call other services or interact with the Eureka server at http://localhost:8070/eureka/, if needed.

So we are trying to create a property which **is eureka.instance.prefer IP address** as true.

The reason why I'm setting this prefer IP address as true is whenever my accounts microservice is trying

to register with the Eureka server by default, it will try to register with the host name.

The host names will make sense whenever you are trying to use DNS mappings inside your microservices network.

But inside our local system, we don't have any DNS mapping set up.

We should use prefer IP address as true so that my accounts microservice will register with my Eureka

server by using the IP address. So behind the scenes, whenever my other microservice, they are trying to connect with the accounts microservice. My Eureka server is going to share the IP address details of accounts microservice so that the other microservices, they can connect with my accounts

microservice. So that's the purpose of this property.Once we mention these properties, now my accounts microservice can connect with my Eureka server. Apart from these properties,

info:  
 app:  
 name: "accounts"  
 description: "MS Bank Accounts Applications"  
 version: "1.0.0"

**Purpose of the info Section**

The info section is a **custom metadata block** often used to provide general information about the application, such as its name, description, and version. Spring Boot uses this information primarily for documentation and monitoring purposes.

**Breakdown of Your Configuration**

1. **info.app.name: "accounts"**
   * Defines the name of the application.
   * In this case, the application is named **"accounts"**, indicating that it is part of a microservices system managing bank accounts.
2. **info.app.description: "MS Bank Accounts Applications"**
   * Provides a short description of what the application does.
   * Here, it describes that the application manages **bank accounts** and is part of a **microservices (MS)** architecture.
3. **info.app.version: "1.0.0"**
   * Specifies the version of the application.
   * This versioning helps in identifying the build or release version of the service.

**Where This Is Useful**

1. **Actuator Endpoints**:
   * If Spring Boot Actuator is enabled in the application, the info metadata is exposed via the /actuator/info endpoint.
   * Example: Accessing http://localhost:8080/actuator/info will return:

json

Copy code

{

"app": {

"name": "accounts",

"description": "MS Bank Accounts Applications",

"version": "1.0.0"

}

}

1. **Service Discovery**:
   * The metadata can be helpful for other services or developers to understand what the application does.
2. **Monitoring and Debugging**:
   * Tools like **Spring Boot Admin** or **Grafana dashboards** can display this metadata to help identify services by their name, description, and version.
3. **Documentation**:
   * This block serves as in-code documentation, making it clear what the application is about, especially in larger teams or multi-service architectures.

endpoint:  
 shutdown:  
 enabled: true  
 info:  
 env:  
 enabled: true  
  
  
  
endpoints:  
 shutdown:  
 enabled: true

This Is Enabling the endpoint in the actuator

### ****Explanation of Each Line****

#### ****1.**** endpoint.shutdown.enabled: true

* **Purpose**:  
  This enables the **shutdown endpoint** in Spring Boot Actuator.
  + The **shutdown endpoint** allows the application to be stopped gracefully by sending a request to a specific URL (e.g., http://localhost:8080/actuator/shutdown).
* **Default Behavior**:  
  By default, this endpoint is disabled for security reasons because it can be misused to bring down an application.
* **Use Case**:  
  It is typically used in **development** or **controlled environments** (e.g., during CI/CD pipeline execution or scripted application shutdown).

#### ****2.**** info.env.enabled: true

* **Purpose**:  
  This enables the info.env property, which is part of the **/actuator/info** endpoint.
  + When this is set to true, additional environment-specific details (like environment variables) can be displayed via the /actuator/info endpoint.
* **Example**:  
  Accessing http://localhost:8080/actuator/info might show:

{

"app": {

"name": "accounts",

"description": "MS Bank Accounts Applications",

"version": "1.0.0"

},

"env": {

"spring.profiles.active": "dev"

}

}

* **Use Case**:  
  This is helpful for debugging and identifying the environment (e.g., dev, test, prod) in which the application is running.

#### ****3.**** endpoints.shutdown.enabled: true

* **Purpose**:  
  This is another way to enable the **shutdown endpoint**, similar to endpoint.shutdown.enabled: true.
  + The difference lies in the configuration structure:
    - **endpoint.shutdown.enabled** is specific to the shutdown endpoint.
    - **endpoints.shutdown.enabled** falls under a broader endpoints block, where multiple endpoints can be configured.
  + Both achieve the same result, but **endpoints.shutdown.enabled** is more consistent in scenarios where you manage multiple endpoints.

### ****Summary of Behavior****

When this configuration is active:

1. The **shutdown endpoint** (/actuator/shutdown) will be available, allowing you to gracefully stop the application.
2. The /actuator/info endpoint will provide environment-specific information, like the active Spring profile or environment variables.

### ****Security Considerations****

* **Shutdown Endpoint**:
  + Should never be enabled in production without proper access control, as it can allow unauthorized users to stop the application.
  + Use Spring Security to restrict access to this endpoint.
* **Info Endpoint**:
  + Ensure sensitive environment variables (like secrets or API keys) are not exposed inadvertently. Mask or disable such data if necessary.

So we have made all the required changes.Now I can save these changes and do a build.

Once the build is completed, I can try to start my accounts microservice. So for the same I'm going

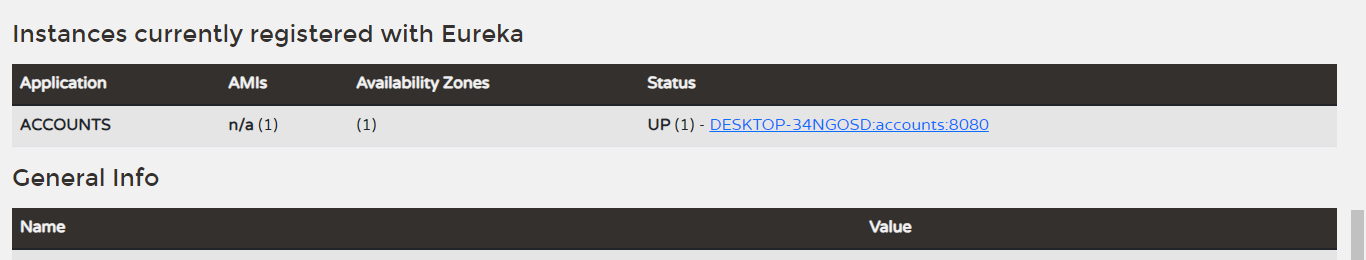
to open the accounts application class and try to start the application in debug mode. But please note that before you try to start the accounts microservice, please make sure config server and Eureka server is started inside your local system. Now here you can see my accounts microservice started at the 8080 and immediately it registered itself with the Eureka Server. That's why you're able to see these registration status as 204.

That means it tried to register with Eureka Server and it got a response 204 from the Eureka Server.

You can also see other kind of information like starting Heartbeat Executor renewal interval is 30s,

which means my Eureka server by default, it is going to expect a heartbeat every 30s from my accounts

microservice. Now let me go to the Eureka dashboard and confirm if my accounts microservice is registered with the Eureka server. The URL that we need to access here is, localhost 8070 which will display the Eureka dashboard.



You can see here right now, under the instances currently registered with Eureka, there is an application with the name accounts. So whatever name you have defined inside the **application.yml** file under the property which is spring.application name, the same name is going to be considered and registered with the Eureka Server. So as of now we have only one instance. That's why you're able to see only one here in the Eureka dashboard and the status of these one instance of accounts

microservice is up. If you try to click on this link, it will show you the more details about these accounts

microservice like what is the app name, what is the description and what is the version?

Now I'm assuming you are clear on how the info related information that we have configured inside the

**application.yml**. The same is linked inside our Eureka dashboard and whenever we click on this you can see against the accounts microservice instance it is trying to invoke **actuator/info** path at the port 8080 and

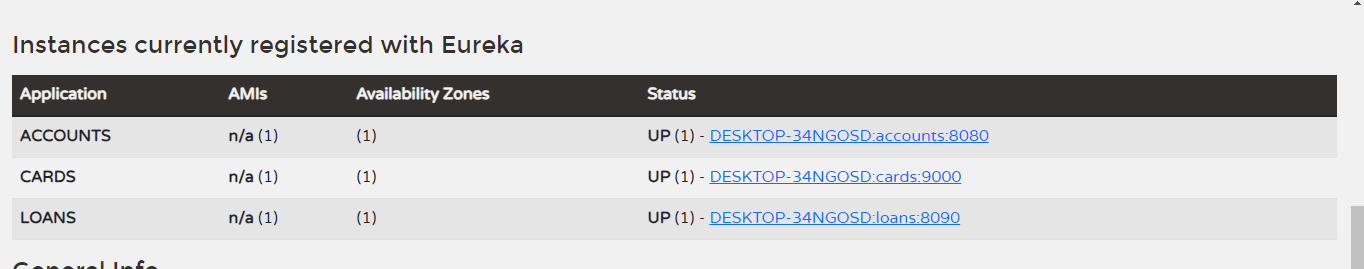
since we registered our accounts microservice using IP address it is using my local IP address. But here you can see it is showing some different hostname. This is because inside our local system, whenever we install Docker and other type of softwares, they're going to make some host entries against the local host IP address.

So that's why instead of local host it is showing some other hostname. So don't worry about this host name now here you see, we are able to see an error, which is like emergency eureka may be incorrectly claiming the instances, so there is some error. Don't worry about this error. We are going to discuss about this beautiful concept why we are seeing this error in the coming lectures. But for next few lectures, please ignore this error.

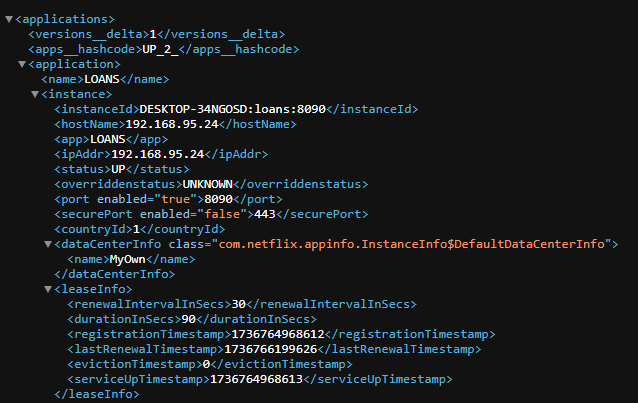
This is not an error. It's a warning about the Eureka dashboard. So with this we are able to successfully connect our accounts Microservice to the Eureka server with that. Now my accounts microservice.

During the startup it resisted itself with the Eureka server and at the same time it is also going to

send the heartbeat signal every 30s. I hope you are clear about all the changes that we have done inside this accounts microservice   
for other microservices loans, cards we will follow same process.. to register the microservices with ths eureka server



<http://localhost:8070/eureka/apps:-> Using this url we can see all the app which are register with our eureka ser ver



If you try to access this, you can see you should be able to see all the apps and the instances that

are registered with the Eureka Server. So under application you can see the application name is loans.

So under this loans, as of now, we have only one instance registered.

If you have multiple instances of loans microservice registered, you will be able to see multiple instances

information. So as of now we are able to see only one instance information.

So this is the instance ID, this is the hostname, this is the app name, IP address,

what is the status and what is the port number? And if you can scroll down, there is also some good information on what is the default renewal interval in seconds and what is the registration time, what is the last renewal timestamp and what is the service up timestamp? And very similarly, what is the home page URL, what is the status page URL? What is the health check? This way we have lot of information about our instance that is registered with the Eureka Server. Very similarly, we also have here accounts related information and this application name accounts. We have one instance registered and the similar lines. You will also see cards microservice and its instance details.

So if you have multiple instances under a microservice, you should be able to see all of them inside

the response whenever someone is trying to invoke this URL. Using this URL, only anyone can connect with my Eureka server and they can try to get the service registration details. As of now, you can see by default the response is coming in an XML format. But what if you want in Json format?

So the Eureka Server supports the Json format also. So to get the response in a Json format, please take this URL and go to the postman..if you click on this request. **http://localhost:8070/eureka/apps/**

So this is the same URL that we tried to invoke before you try to send this request, please make sure you are creating a new header with the name, accept and inside the value. Please make sure you are mentioning application slash Json.

So if you try to uncheck this you can see by default I get in an XML format, but if I try to send this

header with the value application slash Json, I am getting all the response in a Json format.

For some reason,

if you are looking for only **accounts** related information, then you can just mention

accounts inside the URL http://localhost:8070/eureka/apps/accounts

and you will get only accounts related information. And similarly, if you are looking for parts, then mention the same cards inside the URL, you will get only cards related information along with the instance details of cards microservice.

I hope you are seeing and realizing the power of Eureka Server, which is acting as a discovery agent.

So we have only three microservices here. But think of if you have 100 microservice, how useful this Eureka server for any internal communication where one microservice is trying to connect with other microservice. We are not doing any manual job of updating the IP address details into the registry.

All of these details are getting registered automatically during the startup of the microservices.

I hope you are successful at creating an Eureka server and registering all your microservice details

with the Eureka Server during the startup of the microservices.

**De Register From the Eureka Server When microservice shutdown**

As of now, we saw the demo that our individual microservice, they are able to connect with the Eureka

Server during the startup and register themselves.That's why we are able to see the details of our microservices inside the dashboard of Eureka Server. Now, inside this lecture, I'm going to show you a demo of how our microservices, they are going to deregister themselves during the shutdown process. In order to see the same,

we should not shut down our applications or microservices using IntelliJ idea. So here we have an option to shut down our stop our microservices. But we should not use this because this is going to kill your application instantly. And usually we will not follow this approach inside higher environments like dev, qa and prod.

We are going to shut down our applications, our microservices using some scripts or by using the shutdown

path available inside the actuator. Previously, we added some shutdown related properties inside loans, cards and accounts microservice.

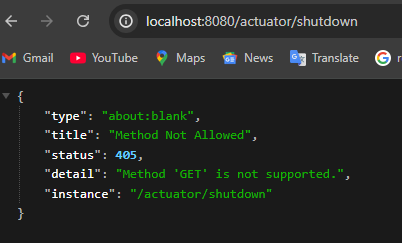
If you can scroll up here, we have enabled the shutdown related endpoints and at the same time, we

also mentioned these properties under management.

So with these properties now there will be an shutdown endpoint exposed inside the actuator of your

microservices. To know more details about the shutdown path available inside the actuator.

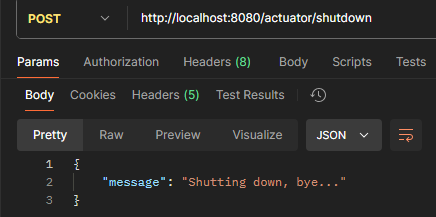
I'm going to invoke this path which is localhost:8080/actuator/shutdown. So if you try to invoke this, you can see here right now it is throwing an error saying that we cannot invoke this API with the help of Http get().



We should use only **http post().** So that's why let's take this URL and try to invoke this API path from our postman. you can see this one we are trying to invoke with the help of post method and this is the URL.

So if I try to send the request to this API path by clicking on this send button, my account service

will try to shut down itself and the shutdown process is not going to be immediate, It is going to take some grace period and make sure it is performing some task before it is trying to shut down. One of such task is to deregister the details with the Eureka Server. So let's try to see this in the demo. I'm clicking on this send button. You can see I got a message saying that shutting down, Bye bye. So this is our response that we got.



Now let's go to the Eureka dashboard and see if our accounts microservice details are available. So here I will try to refresh this page.You can see here right now, AccountsApplication related information is not present inside this Eureka dashboard.  


That means my accounts microservice during the shutdown process, it deregistered itself from the Eureka server. And with that, there won't be any details about accounts microservice inside the service registry that

is maintained by the Eureka Server. Inside the logs, we can also see this. If we can open the logs related to AccountsApplication, you should be able to see some logs here. You can see here stopping service and there is a log related to Unregistering and there is a message also. So and so account is de-registered and the status from the Eureka server is 200 confirming that all the details about these accounts microservice instance is removed from the service registry.

**Demo of heartbeats mechanism to Eureka Server From Client**

Now inside this lecture, let's see the demo of heartbeats that will be sent by the microservices to

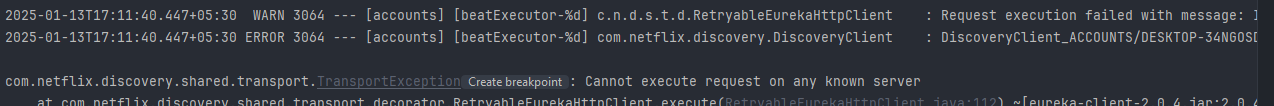
the Eureka Server. So to get started with this discussion, first, I need to make sure all of my microservices are started. So let me start all my microservices. So by the time these microservices starts successfully. Behind the scenes, they will also register their instance details with the Eureka Server. So we can also validate the same by going into the Eureka dashboard. So here I'm going to refresh this page and you can see this time, we are able to see all the instances currently registered with Eureka.



**As a next step** to show you that microservices are trying to attempt to send the heartbeat every 30s, which is the default period. I'm going to shut down the Eureka server. Here before I try to shut down the Eureka Server.

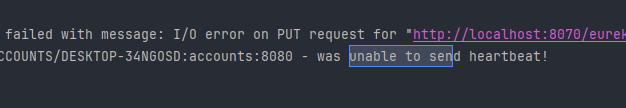
Let me clean all the console of accounts, loans and cards microservice. Post that I'm going to stop the **Eureka Server**. With that, we will be able to see some exceptions inside the console of our microservices. That they

are trying to send the heartbeat but Eureka Server is not responding. So let's wait for 30s. Post that we should be able to see some errors inside the microservices console.



So here you can see for AccountsApplication, a new exception is generated. If you go and try to understand these exception, there will be details related to heartbeat. Maybe what we can do is, we can try to search for a word heartbeat here. So let me search for this

and there are so many matches with the word heartbeat, you should be able to see these kind of message like was unable to send Heartbeat to Eureka Server.



You can see in the logs it is trying to send the request of type PUT to so and so URL of Eureka to update the heartbeat details. But since the Eureka server itself is down, we are seeing this exception inside the console of AccountsApplication.The same we can see inside the LoansApplication also and followed by CardsApplication even inside the CardsApplication console also we have errors related to heartbeat.

So this confirms that our individual microservices, they are trying to send the heartbeat signal to the Eureka server. With this it should be clear for you, like during the startup, the registration will be taken care

by the individual microservices and once the services are started, even the heartbeat also will be

sent every 30s to the Eureka server to make sure that Eureka server is maintaining only the instance

details which are healthy and at last during the shutdown process also our microservices, they are going to unregister themselves and all these steps are going to happen

automatically. No where we are doing this manually, the communication between microservices and Eureka Server is happening automatically. So now we have the Eureka Server, which is acting as a service discovery agent and at the same time our microservices are also communicating their instance details to the Eureka Service Registry.

As a next step, we need to understand how a microservice can rely on this Eureka server to discover

the other service details and in the same process how it is going to perform the load balancing.

**Feing Client Code changes to invoke other microservices Part1**

As a next step, let's try to explore more about service discovery and how the client side load balancing

is going to work whenever we are using Eureka Server inside our microservices network.

To show a demo about these concepts, we are going to build a new REST API inside the accounts microservice. The responsibility of these new REST API is, it has to accumulate account related information, loans related information and cards related information based upon a given mobile number and it should consolidate all the response from the other microservices like loans and cards along with the accounts related data. So all the data of a given customer, it has to respond back to the client whoever is invoking these new REST API.

So here you can see we have a scenario where my accounts microservice does not have any information related to cards and loans. So it has to connect with the cards and loans microservice to get that data. And that's how we are going to have a scenario where accounts microservice is trying to communicate with the other microservices internally inside our microservice network**. So to establish the internal communication between the microservices like we discussed previously, we need to use open feign client library.**to use feign client

**Step 1** :- add this dependency inside the pom.xml

<dependency>

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

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

</dependency>

**Step 2** :- Add this annotation inside the main application calss

@EnableFeignClients

This will provide the all the functionality related to feing client

So the purpose on why we need to mention this annotation is this will enable the feign clients related

functionality inside my accounts microservice. With that my account microservice can connect with other microservices like loans and cards. So let me save this changes. Now here in order to connect with the loans and cards microservice in a traditional approach, we will use some rest template or web client and to those REST template and web client classes are interfaces. We are going to pass,

what is the URL,

what is the request data,

what is the port number,

So we need to write all the implementation code or actual logic to invoke the loans and cards microservice.

But whenever we are using open **feign client**, we don't have to write the implementation code.

We just have to write that declarative code. We are already following this kind of approach inside the spring data JPA. For suppose if you go to the repository package. Here we have accounts repository.

**findByCustomerId(Long CustomerID)** Did I write any implementation code here for to fetch the records based upon the customerId ?

No, I have written only that declarative code, which means I only created an interface along with the abstract methods and behind the scenes my spring data JPA framework is going to generate the implementation code at the runtime.

Very similarly, whenever we are trying to use open feign client, we don't have to write the implementation

code.

We just need to create the interfaces along with the abstract methods. So let's try to understand how to develop the same. For the same here, just under the services, I'm going to create a new package with the name com.msbytes.accounts.service.client under this client package

*@FeignClient*("cards")  
*public interface* CardsFeignClient {  
}

So let me create a new interface here. So the interface name, I'm going to keep it as CardsFeignClient because this interface is going to help my accounts microservice, to connect with the cards microservice. So let me create this new interface. So now I have an empty interface. To use the open Feign libraries,

we need to make sure we are mentioning an annotation here which is **@FeignClient**. To this @FeignClient, :- We need to mention a value which is cards because the same value, which is cards we have used to register with the Eureka Server. If you can recall inside the Eureka dashboard, all the cards related instances, they will register with the logical name cards. So the same cards we need to mention here so that my **feign client** will connect with the Eureka server at the runtime and it will try to get all the instance details with the logical name cards. So that's the purpose of this annotation and the value that we have mentioned here.

***@FeignClient*("cards"):-** This Will allow to interact with the cards microservice and we can get all the instance details of card microservice from the Eureka server

**Step 3** :- Now, inside this interface, I'm going to create a new abstract method and whatever abstract method that we are going to create inside this interface, this method signature has to match with the actual REST API method that we have defined inside the card microservice.

If you go to the cards controller, inside the cards controller, we have a method with the name fetch

card details.So I need to create the very similar method inside my CardsFeignclient interface. So let me copy this method details and mention the same here. But here inside this interface we are not going to write any implementation code. We are just going to define them as abstract methods. Once we define these abstract method, we can remove these validations related code because the validations will be performed at the REST API level but not here. That's why we don't have to mention the validations.

Now you can see here we are following the same method name like we have inside the CardsController.

The method name you can choose whatever name inside this FeignClient interface, but please make sure the method signature like method input parameters, return parameters along with the method access type should be same as what we have defined inside the actual microservice.

So once we define this here, you can see we don't have the CardsDto inside the accounts microservic.

So let's try to create the CardsDto inside the accounts microservice. So for the same, I'm going to copy the DTO from the cards microservice. So if I go to the cards microservice I'll be having a package and the dto package I need to copy the CardsDto class. So let me copy the same into the dto package of accounts microservice.

*@FeignClient*("cards")  
*public interface* CardsFeignClient {  
 *@GetMapping*(value="api/fetch",consumes = "application/json")  
 *public* ResponseEntity<CardsDto> fetchCardDetails(*@RequestParam* String mobileNumber);  
}

if you see inside the CardsController, we also have some path which is specific to entire controller. The path is **/api.** So that's why we need to mention the exact path, which is **/api/fetch**. With this we should be good if needed we can also mention like consumes as **application/json**. We are trying to communicate to the FeignClient that these API which we have inside the cards microservice, it is going to accept that Json input data. So now if you see this abstract method is matching exactly with what we have defined inside the cards microservice. So that's the one primary rule that we need to follow.

On top of that, we need to make sure we are mentioning this annotation along with the logical name

that cards microservice is going to register with the Eureka Server. So behind the scenes, my cards FeignClient will connect with the Eureka Server and try to fetch all the instances that are registered with the logical name cards.And once it receives 1 or 2 or any other instance details, it will try to cache those details for 30s, which is the default period. And within these 30s it is not going to connect again with the Eureka Server, but instead it is going to leverage the details present inside the cache.

So based upon the IP details inside the cache, it is going to invoke this API along with the request which

is mobile number. So behind the scenes, all the implementation code will be generated by the open feign client.

So here you can see we have not written any business logic.

We are just telling to our feign client how to connect to other microservice.

What is the method signature,

what is the request parameter,00

what is the request structure,

what is the response structure,

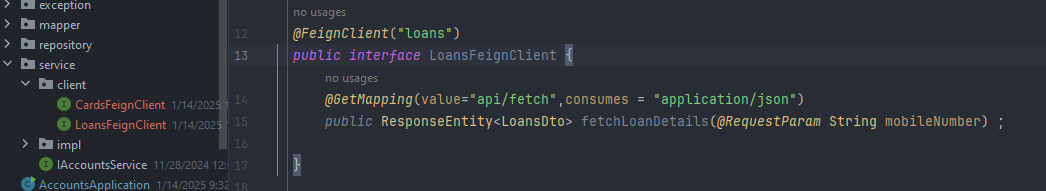
what is the rest API path ?

**So if you define those meta details, the remaining magic will be taken care by the feign client.**

define the abstract method(which we want to access/call from this feing client like we are going to fetch the fetchCardDetails method of cards microservice from this feing client so we are delcalring same fetchcardDetails method for cards microservicse here ) inside this Interface

*@FeignClient*("cards")  
*public interface* CardsFeignClient {  
 *@GetMapping*(value="api/fetch",consumes = "application/json")  
 *public* ResponseEntity<CardsDto> fetchCardDetails(*@RequestParam* String mobileNumber);  
}

Accounts microservice to interact with the other microservices like loans we will follow the same step. I have write the direc code after following all step for loans

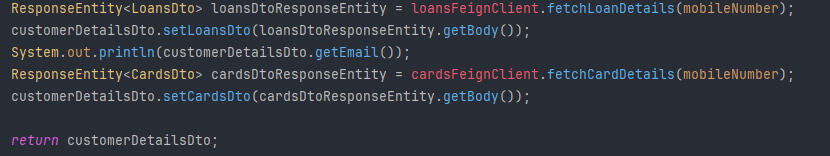


**Feign Client code changes to invoke other microservices-Part 2 (**try to use them inside our accounts microservice to communicate with the other microservices**)**

We will inject the feing client interface inside the service where we want to use the feing



**and after that we can call the microservices**



**In below example we are calling the cards and loans microservices inside the accounts microservice**

*@Service  
@AllArgsConstructor  
public class* CustomersServiceImpl *implements* ICustomersService {  
  
 *private* AccountsRepository accountsRepository;  
 *private* CustomerRepository customerRepository;  
 *private* CardsFeignClient cardsFeignClient;  
 *private* LoansFeignClient loansFeignClient;  
  
 */\*\*  
 \* @param mobileNumber - Input Mobile Number  
 \* @return Customer Details based on a given mobileNumber  
 \*/  
 @Override  
 public* CustomerDetailsDto fetchCustomerDetails(String mobileNumber) {  
 Customer customer = customerRepository.findByMobileNumber(mobileNumber).orElseThrow(  
 () -> *new* ResourceNotFoundException("Customer", "mobileNumber", mobileNumber)  
 );  
 Accounts accounts = accountsRepository.findByCustomerId(customer.getCustomerId()).orElseThrow(  
 () -> *new* ResourceNotFoundException("Account", "customerId", customer.getCustomerId().toString())  
 );  
  
 CustomerDetailsDto customerDetailsDto = CustomerMapper.mapToCustomerDetailsDto(customer, *new* CustomerDetailsDto());  
 customerDetailsDto.setAccountDto(AccountMapper.mapToAccountsDto(accounts, *new* AccountDto()));  
 System.out.println("Hello Testing");  
 ResponseEntity<LoansDto> loansDtoResponseEntity = loansFeignClient.fetchLoanDetails(mobileNumber);  
 customerDetailsDto.setLoansDto(loansDtoResponseEntity.getBody());  
 System.out.println(customerDetailsDto.getEmail());  
 ResponseEntity<CardsDto> cardsDtoResponseEntity = cardsFeignClient.fetchCardDetails(mobileNumber);  
 customerDetailsDto.setCardsDto(cardsDtoResponseEntity.getBody());  
  
 *return* customerDetailsDto;  
  
 }  
}