**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.

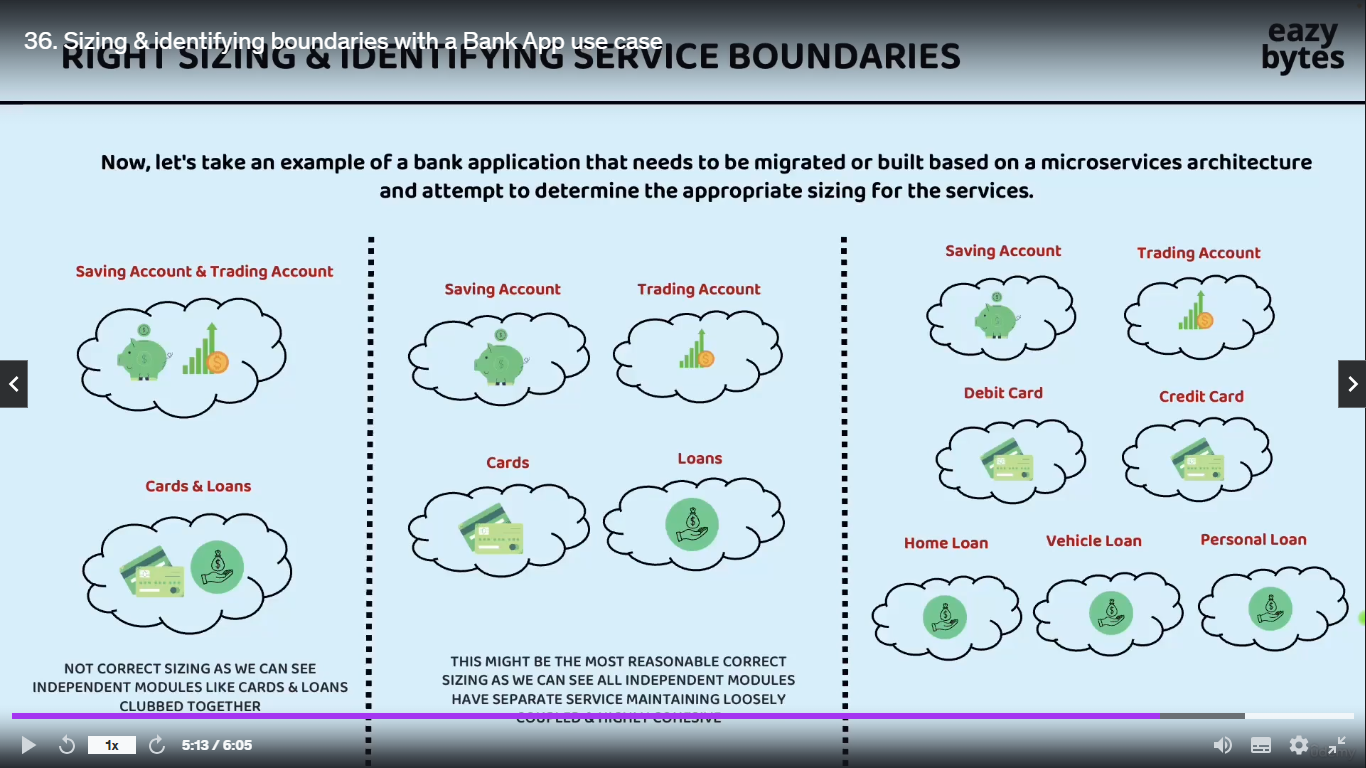
**Payment Processing**: Manages payment-related events.

**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**

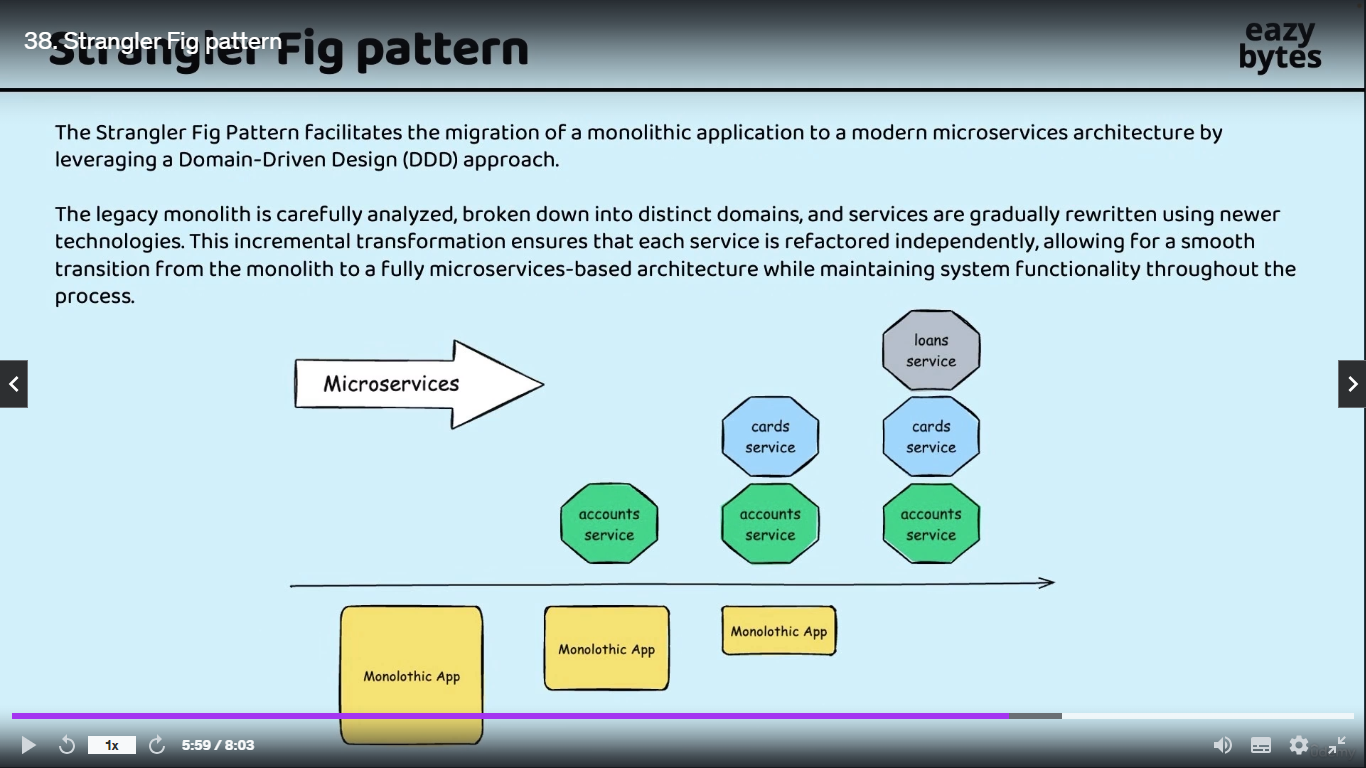
**Team Three**

**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**

