eazy
bytes

MASTER

MICROSERVICES

WITH SPRINGBOOT, DOCKER, KUBERNETES

COURSE AGENDA

Welcome to the world of Microservices

Building microservices business logic using Spring Boot

How do we right size our microservices & identify boundaries

How to containerize our microservices using Docker

COURSE AGENDA

Configurations Management in microservices using Spring Cloud Config

Service Discovery & Service Registration in microservices using Eureka

Building an edge server for microservices using Spring Cloud Gateway

Making Microservices Resilient using Resiliency4J patterns

COURSE AGENDA

Observability and monitoring of microservices using Grafana, Prometheus etc.

Securing microservices using OAuth2/OpenID, Spring Security

Event Driven microservices using RabbitMQ,Spring Cloud Functions & Stream

Event Driven microservices using Kafka,Spring Cloud Functions & Stream

COURSE AGENDA

Container Orchestration using Kubernetes

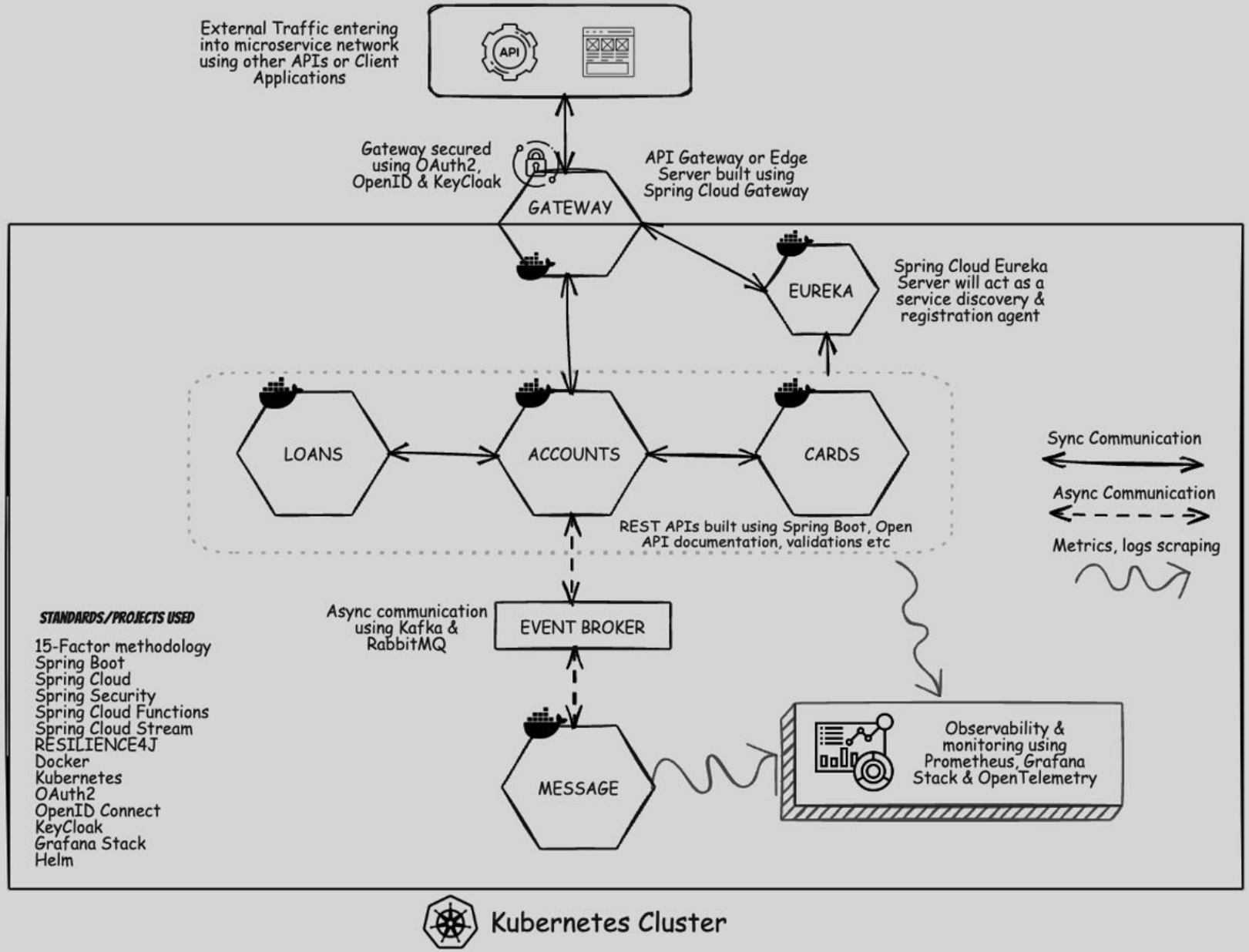
Deep dive on Helm (kubernetes package manager)

Deploying microservices into cloud kubernetes cluster

Many best practices, techiniques followed by real time microservice developers

WHAT WE ARE GOING TO BUILD IN THIS COURSE ?

eazy bytes



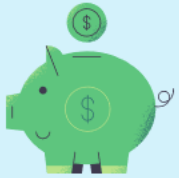
What are Microservices ?

eazy bytes

To understand microservices, let's imagine a bank called **EazyBank**.

Typically, banks comprise various departments, including **Accounts**, **Cards**, and **Loans**.

Cards

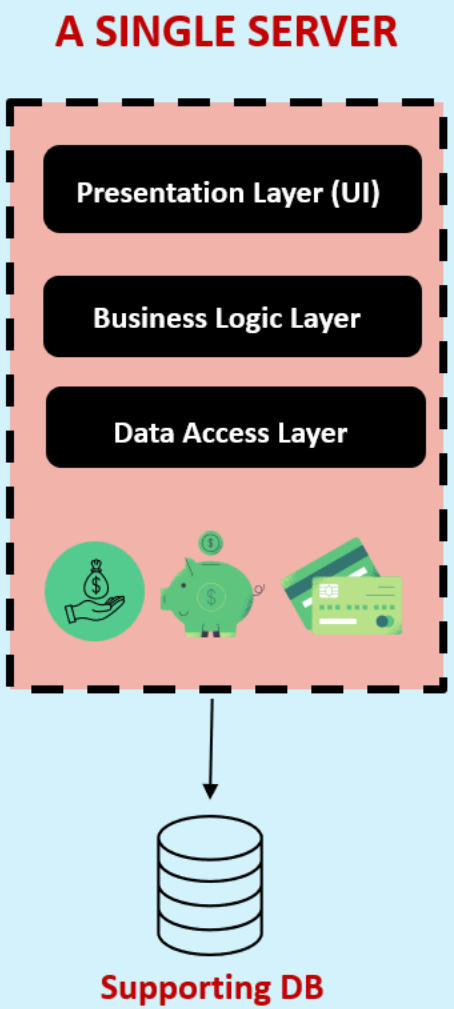


Accounts

Loans

The Monolith

eazy bytes



Back a decade, all the applications used to be deployed as a Single unit where all functionality deployed together inside a single server. We call this architecture approach as Monolithic.

Pros

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

Cons

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

We have various forms of Monolithic with the names like Single-Process Monolith, **Modular Monolith**, **Distributed Monolith**

The Monolith

eazy bytes

Accounts Dev Team

Loans Dev Team

Cards Dev Team

UI/UX Dev Team

Single Code repo

CI/CD

In a monolithic approach, developers work with a single code base, which is then packaged as a unified unit, such as an EAR/WAR file, and deployed onto a single web/application server. Additionally, the entire application is supported by a single database.

Application/Web Server (Tomcat, Jboss, Weblogic, Websphere)

UI

Spring MVC

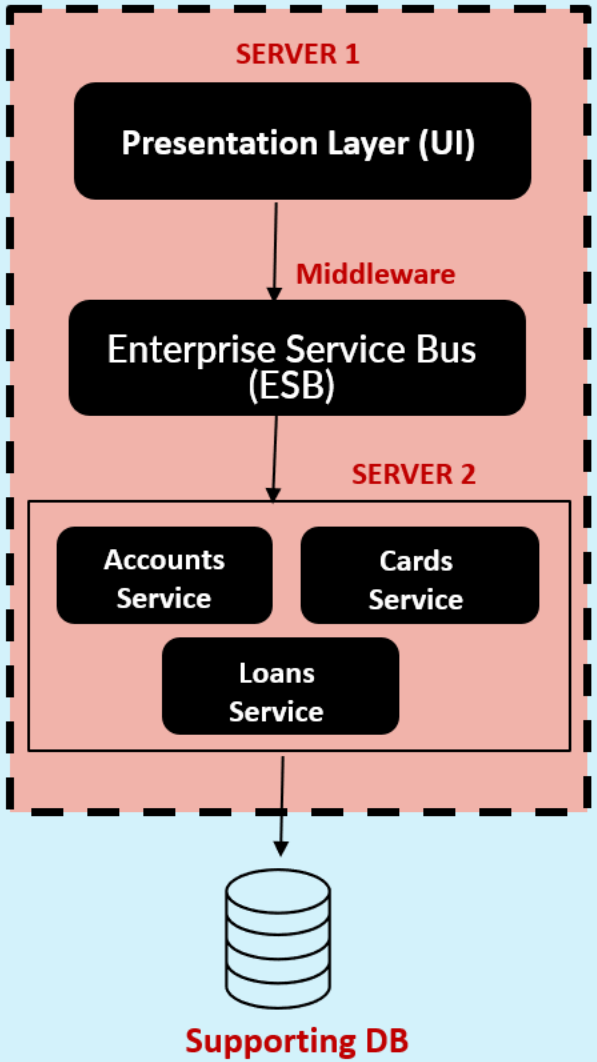
Spring Data JPA/ORM

Single EAR/WAR deployed

Supporting DB

The SOA (Service-Oriented Architecture)

eazy bytes



SOA emerged as an approach to combat the challenges of large, monolithic applications. It is an architectural style that focuses on organizing software systems as a collection of loosely coupled, interoperable services. It provides a way to design and develop large-scale applications by decomposing them into smaller, modular services that can be independently developed, deployed, and managed.

Pros

Reusability of services
Better maintainability
Higher reliability
Parallel development

Cons

Complex management due to communication protocols (e.g., SOAP)
High investment costs due to vendor in middleware
Extra overload

The SOA (Service-Oriented Architecture)

eazy bytes

UI/UX Dev Team

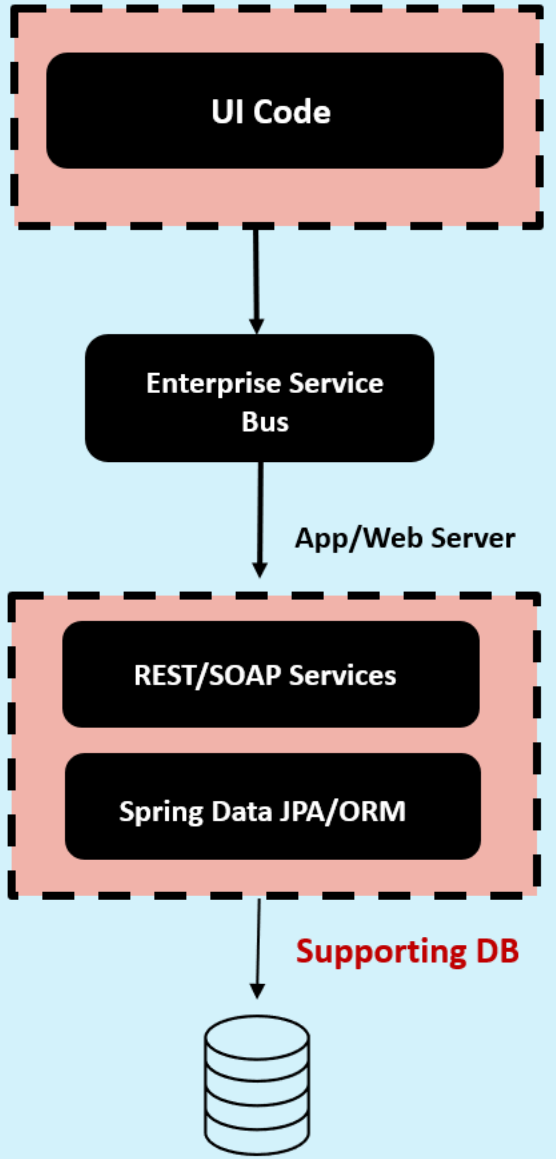
Accounts Dev Team

Loans Dev Team

Cards Dev Team

UI Code repo

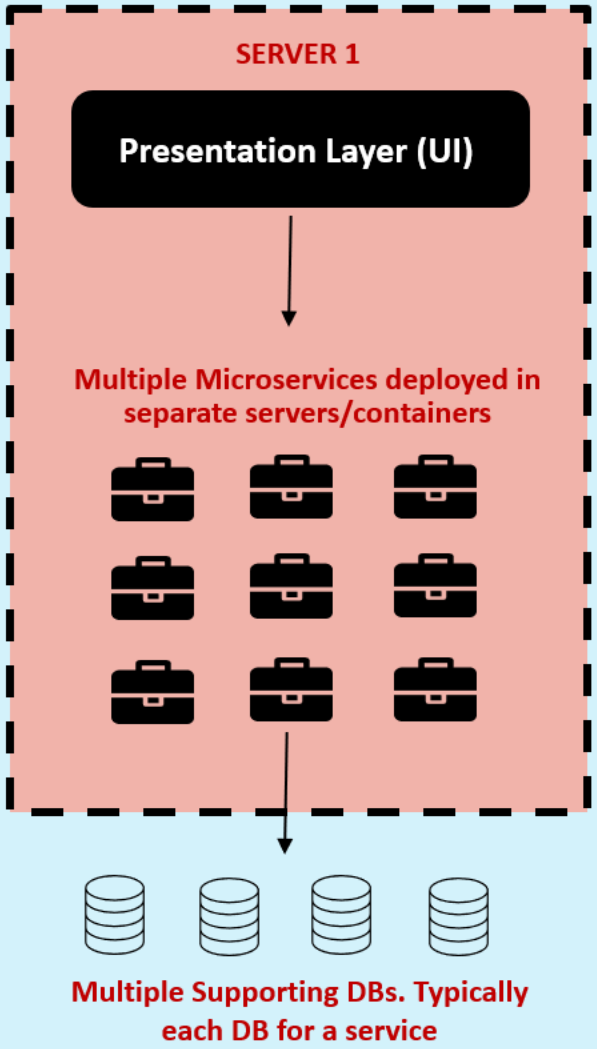
Backend Services repo



I have observed numerous instances of SOA where teams aimed to create smaller services, yet they remained tightly coupled to a shared database and required deploying everything as a cohesive unit. While they followed a service-oriented approach, it cannot be classified as microservices.

The GREAT MICROSERVICES

eazy bytes



Microservices are independently releasable services that are modeled around a business domain. A service encapsulates functionality and makes it accessible to other services via networks—you construct a more complex system from these building blocks. One microservice might represent Accounts, another Cards, and yet another Loans, but together they might constitute an entire bank system.

Pros

Easy to develop, test, and deploy
Increased agility
Ability to scale horizontally
Parallel development
Modeled Around a Business Domain

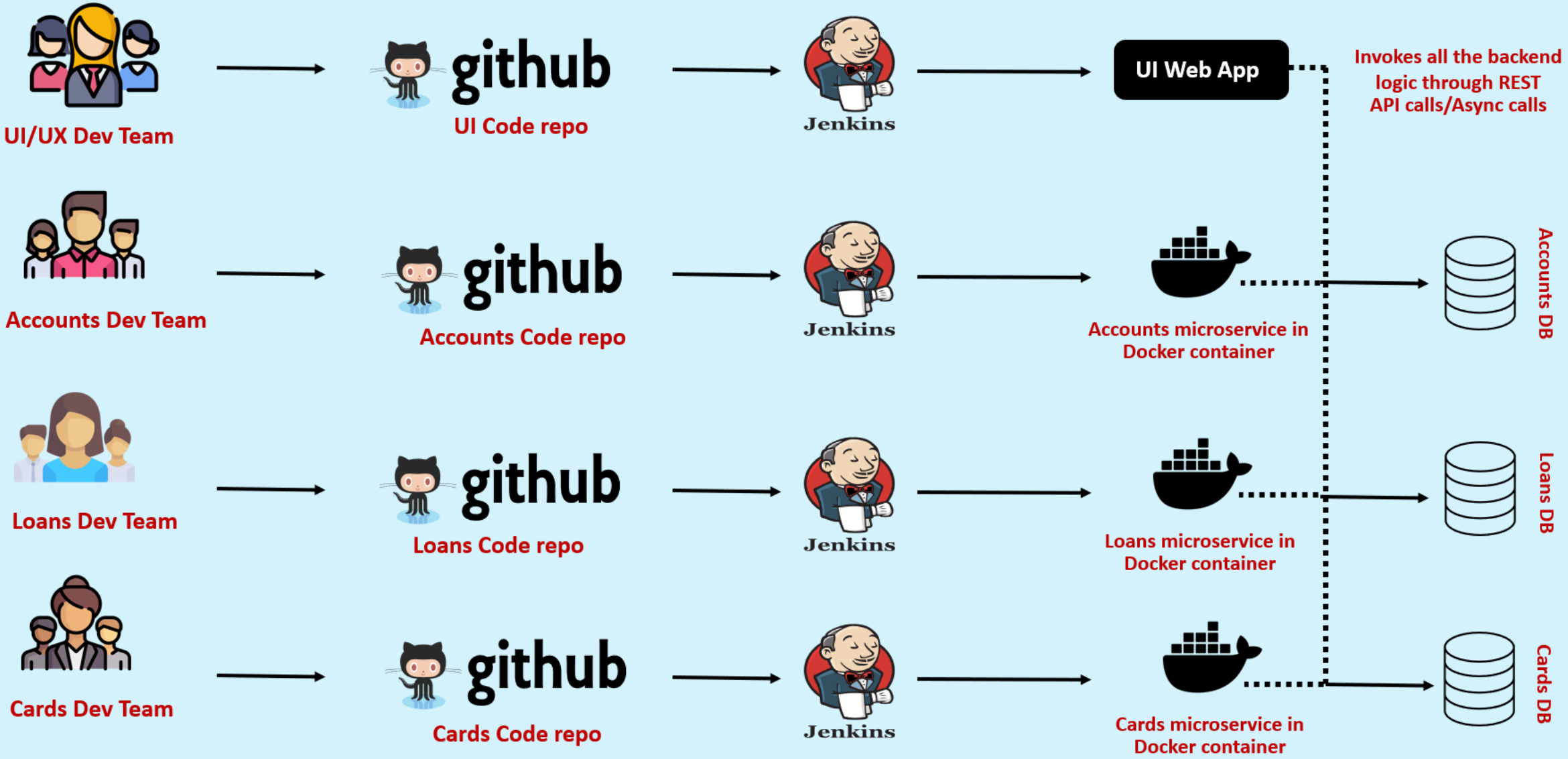
Cons

Complexity
Infrastructure overhead
Security concerns

If there's one crucial takeaway from this course and the concept of microservices, it is to prioritize the independent deployability of your microservices. Develop the habit of deploying and releasing changes to a single microservice in production without requiring the deployment of other components. By doing so, numerous benefits will naturally emerge.

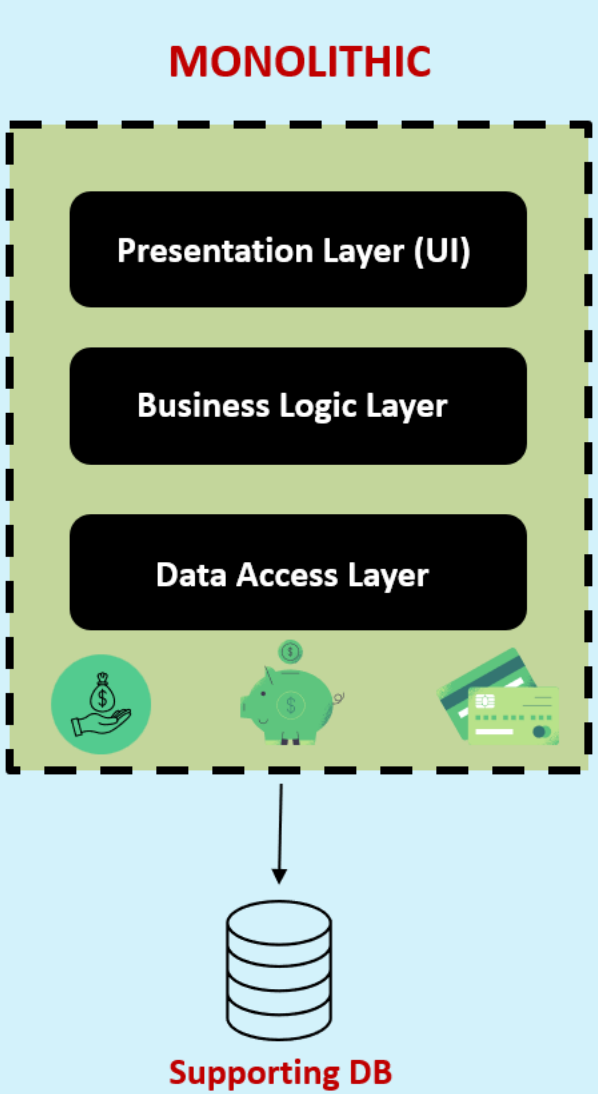
The GREAT MICROSERVICES

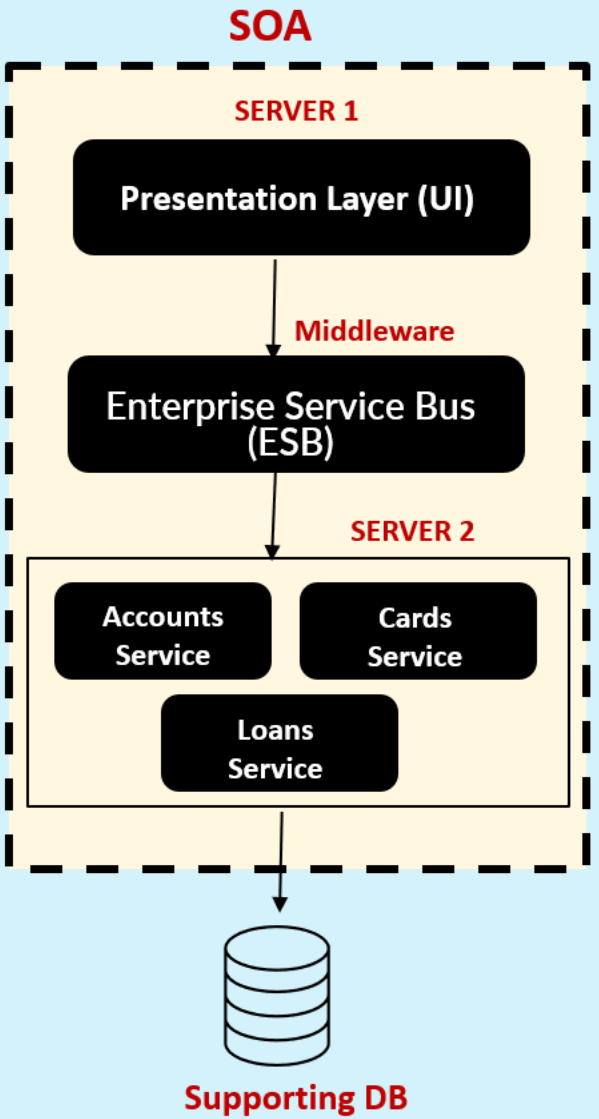
eazy bytes

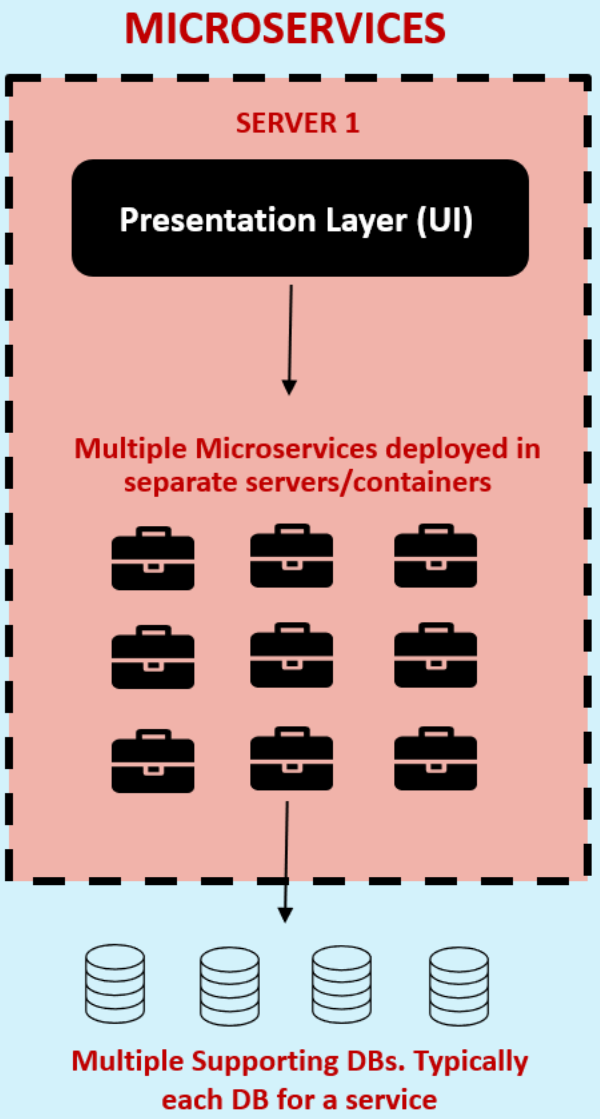


MONOLITHIC vs SOA vs MICROSERVICES

eazy bytes

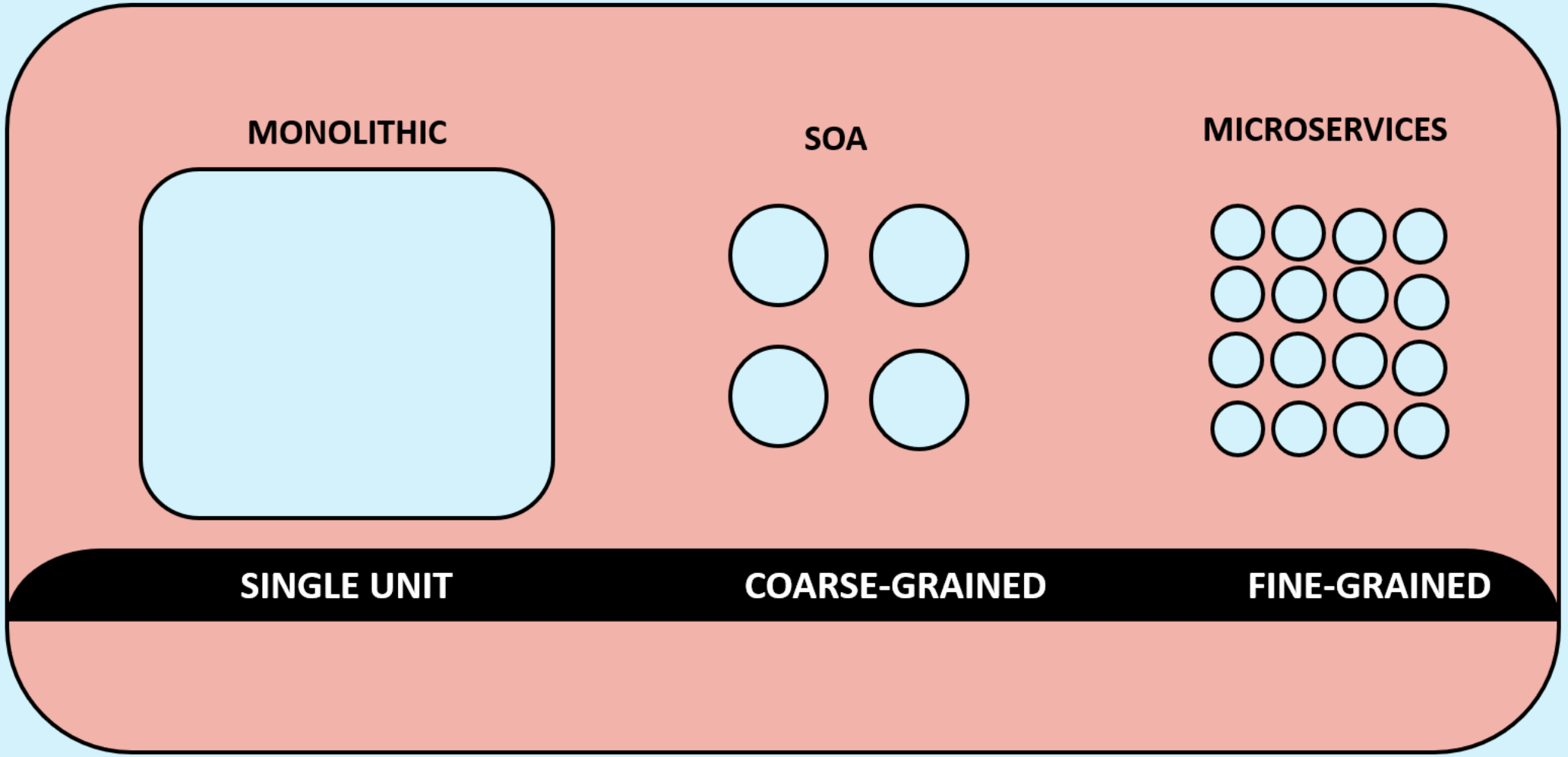






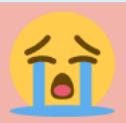
MONOLITHIC vs SOA vs MICROSERVICES

eazy bytes



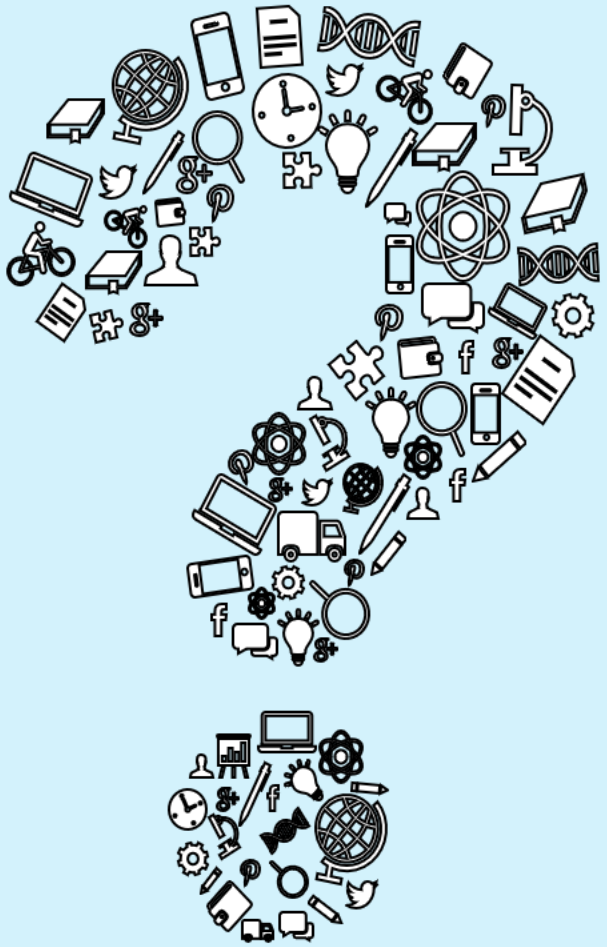
MONOLITHIC VS SOA VS MICROSERVICES

eazy bytes

FEATURES	MONOLITHIC	SOA	MICROSERVICES
Parallel Development			
Agility			
Scalability			
Usability			
Complexity & Operational overhead			
Security Concerns & Performance			

DEFINITION OF MICROSERVICE ?

eazy bytes

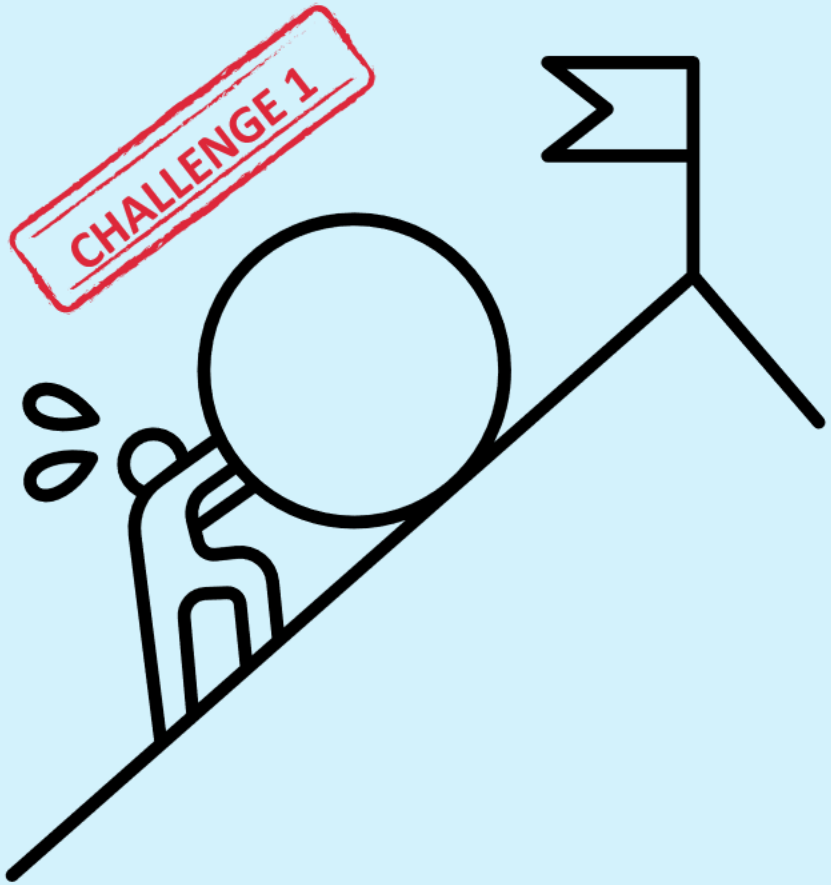


“Microservices is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, built around business capabilities and independently deployable by fully automated deployment machinery.”

- From Article by James Lewis and Martin Fowler’s

HOW TO BUILD MICROSERVICES ?

eazy bytes



when considering a web application, the traditional approach involves packaging it as a WAR or EAR file. These archive formats are commonly used to bundle Java applications, which are then deployed to web servers like Tomcat or application servers like WildFly.

Do you think the same approach works for building microservices ? Off course not, because Organizations may need to build 100s of microservices. Building, packaging, and deploying all the microservices using traditional methods can be an extremely challenging and practically impossible task.

How to overcome this challenge ?

The clue is Spring Boot

WHY SPRING BOOT FOR MICROSERVICES?

eazy bytes

WHY SPRING BOOT IS THE BEST FRAMEWORK TO BUILD MICROSERVICE

Spring Boot is a framework that simplifies the development and deployment of Java applications, including microservices. With Spring Boot, you can build self-contained, executable JAR files instead of the traditional WAR or EAR files. These JAR files contain all the dependencies and configurations required to run the microservice. This approach eliminates the need for external web servers or application servers.

Provides a range of built-in features and integrations such as auto-configuration, dependency injection, and support for various cloud platforms

Provides an embedded Tomcat, Jetty, or Undertow server, which can run the microservice directly without the need for a separate server installation

Inbuilt support of production-ready features such as metrics, health checks, and externalized configuration

We can quickly bootstrap a microservice project and start coding with range of starter dependencies that provide pre-configured settings for various components such as databases, queues etc

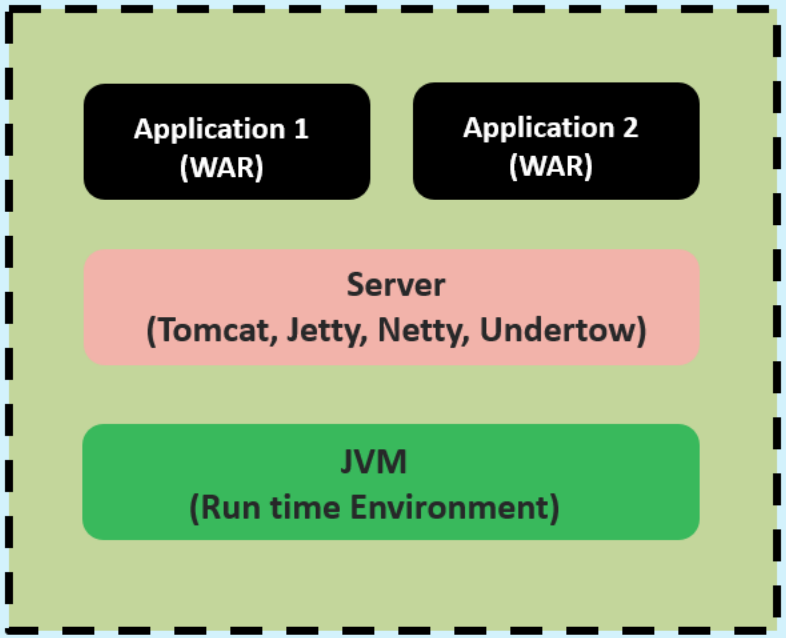
Well-suited for cloud-native development. It integrates smoothly with cloud platforms like Kubernetes, provides support for containerization, and enables seamless deployment to popular cloud providers

WHY SPRING BOOT FOR MICROSERVICES?

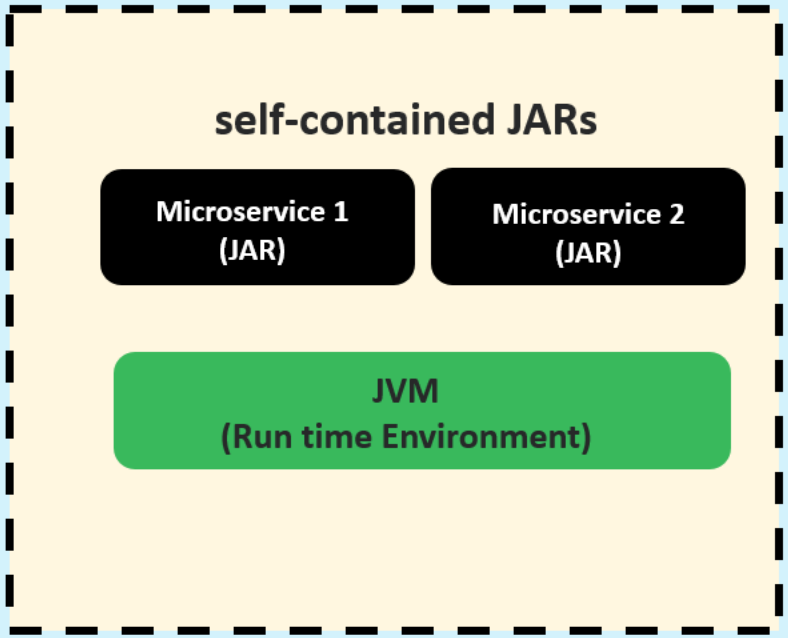
eazy bytes

WHY SPRING BOOT IS THE BEST FRAMEWORK TO BUILD MICROSERVICE

In the traditional approach, applications are typically packaged as WARs and rely on the presence of a server in the execution environment for running. However, in the micro services paradigm, applications are packaged as self-contained JARs, also called fat-JARs or uber-JARs, since they contain the application itself, the dependencies, and the embedded server.



WARs with external server
(Traditional approach)



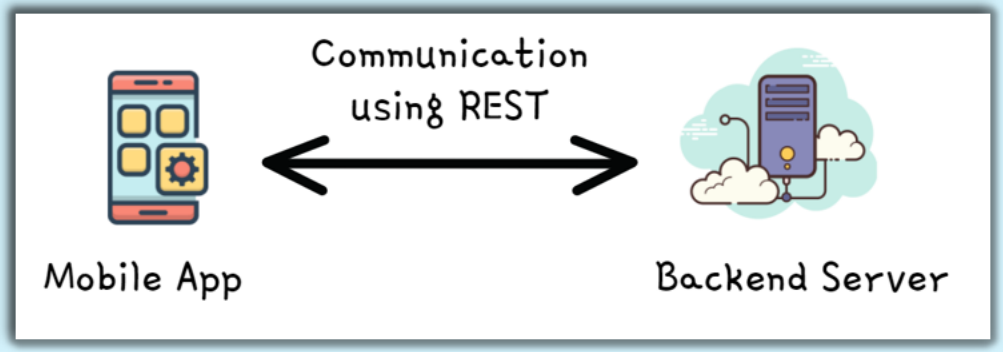
JARs with embedded server
(Modern approach)

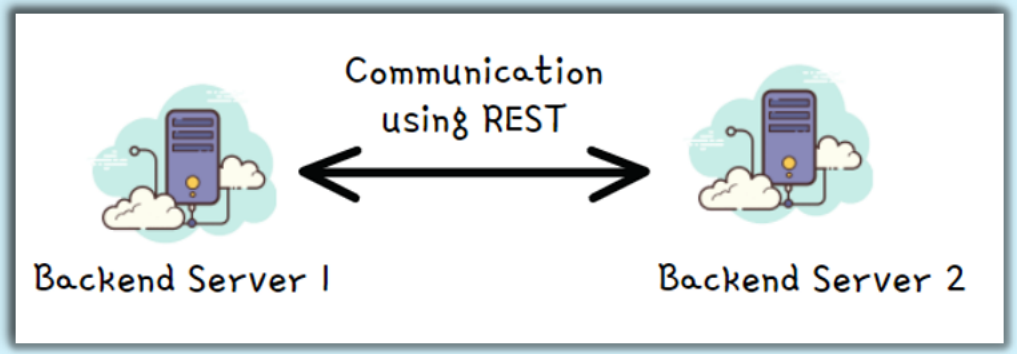
Implementing REST Services

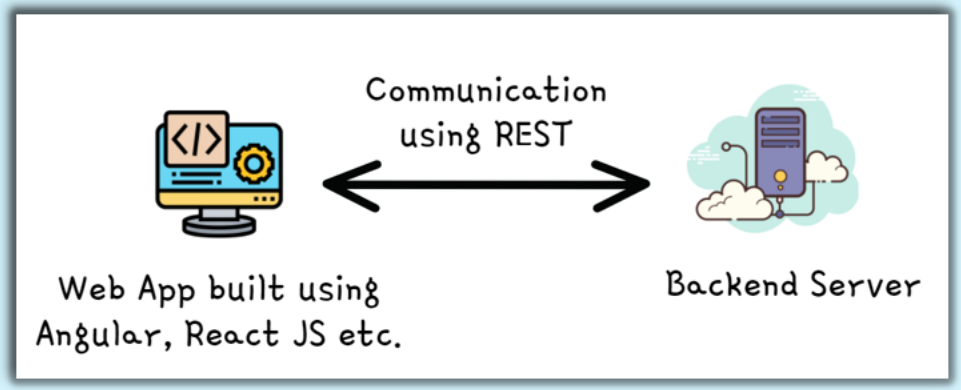
eazy bytes

REST (Representational state transfer) services are one of the most often encountered ways to implement communication between two web apps. REST offers access to functionality the server exposes through endpoints a client can call.

Below are the different use cases where REST services are being used most frequently these days,







Implementing REST Services

eazy bytes

Typically REST services are built to expose the business functionality and support CRUD operations on the storage system. Attached are the standars that we need to follow while building REST services,

Business logic supporting CRUD operations

Create -> HttpMethod.POST
Read -> HttpMethod.GET
Update -> HttpMethod.PUT/PATCH
Delete -> HttpMethod.DELETE

Proper input validation & exception Handling

Make sure all the REST services perform input validations, handle the runtime and business exceptions propertly. In all kind of scenarios, REST services should send a meaningful response to the clients

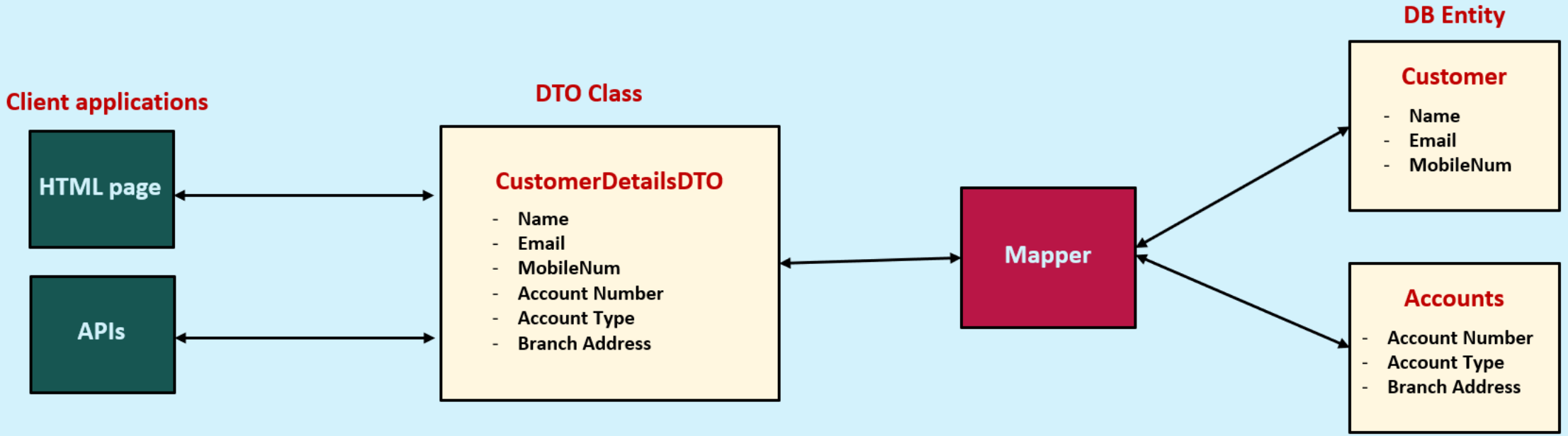
Document REST Services

With the help of standards like Open API Specification, Swagger, make sure to document your REST APIs. This helps Your client, third party developers to understand your services clearly.

DTO(Data Transfer Object) pattern

eazy bytes

The Data Transfer Object (DTO) pattern is a design pattern that allows you to transfer data between different parts of your application. DTOs are simple objects that contain only data, and they do not contain any business logic. This makes them ideal for transferring data between different layers of your application, such as the presentation layer and the data access layer.



Here are some of the benefits of using the DTO pattern:

Reduces network traffic: DTOs can be used to batch up multiple pieces of data into a single object, which can reduce the number of network requests that need to be made. This can improve performance and reduce the load on your servers.

Encapsulates serialization: DTOs can be used to encapsulate the serialization logic for transferring data over the wire. This makes it easier to change the serialization format in the future, without having to make changes to the rest of your application.

Decouples layers: DTOs can be used to decouple the presentation layer from the data access layer. This makes it easier to change the presentation layer without having to change the data access layer.

Different Annotations & Classes that supports building REST services

eazy bytes

@RestController – can be used to put on top of a call. This will expose your methods as REST APIs. Developers can also use @Controller + @ResponseBody for same behaviour

@ResponseBody – can be used on top of a method to build a Rest API when we are using @Controller on top of a Java class

ResponseEntity<T> - Allow developers to send response body, status, and headers on the HTTP response.

@ControllerAdvice – is used to mark the class as a REST controller advice. Along with @ExceptionHandler, this can be used to handle exceptions globally inside app. We have another annotation @RestControllerAdvice which is same as @ControllerAdvice + @ResponseBody

RequestEntity<T> - Allow developers to receive the request body, header in a HTTP request.

@RequestHeader & @RequestBody – is used to receive the request body and header individually.

Summary of the steps followed to build microservices

eazy bytes

01 Build empty Spring Boot applications

First we created empty Spring Boot applications with the required starter dependencies related to web, actuator, JPA, devtools, validations, H2 DB, Lombok, spring doc open API etc.

02 Build DB related logic, entities & DTOs

We created required DB tables schema, established connection details with H2 DB, created JPA related entities, repositories. Post that using DTO pattern guidelines, we built DTO classes and mapper logic inside all the microservices

03 Build business logic

Inside all the microservices, we created REST APIs supporting CRUD operations with the help of various annotations like **@PostMapping**, **@GetMapping**, **@PutMapping**, **@DeleteMapping** etc.

04 Build Global Exception handling logic

We built global exception handling logic using annotations like **@ControllerAdvice** & **@ExceptionHandler**. Also created custom business exceptions like **CustomerAlreadyExistsException**

05 Perform data validations on the input

Perform validations on the input data using annotations present inside the **jakarta.validation** package. These annotations are like **@NotEmpty, @Size, @Email, @Pattern, @Validated, @Valid** etc.

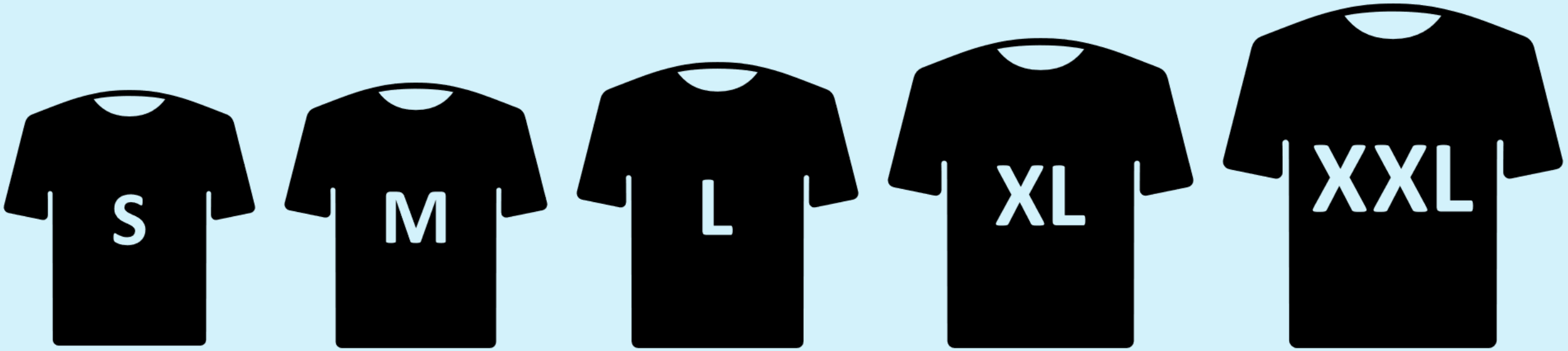
06 Perform auditing using Spring Data JPA

With the help of annotations like **@CreatedDate, @CreatedBy, @LastModifiedDate, @LastModifiedBy, @EntityListeners & @EnableJpaAuditing**, we implemented logic to populate audit columns in DB.

07 Documenting REST APIs

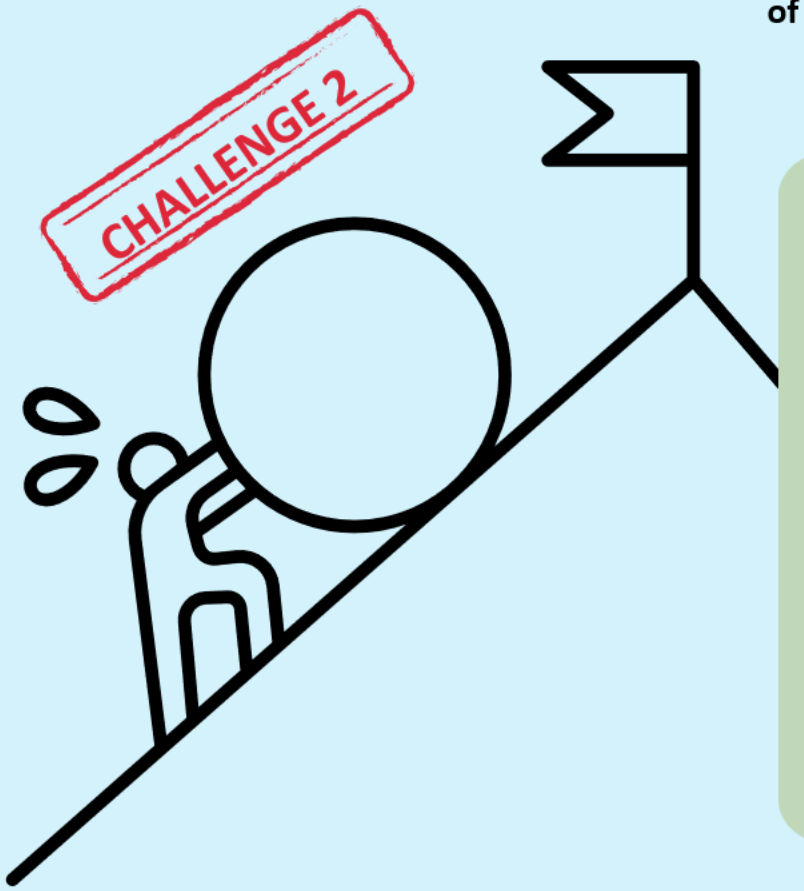
With the help of OpenAPI specifications, Swagger, Spring Doc library, we documented our REST APIs. In the same process, we used annotations like **@Schema, @Tag, @Operation, @ApiResponse** etc.

eazy
bytes



HOW TO RIGHT SIZE & IDENTIFY SERVICE BOUNDARIES OF MICROSERVICES ?

eazy bytes



One of the most challenging aspects of building a successful microservices system is the identification of proper microservice boundaries and defining the size of each microservice. Below are the most common followed approaches in the industry,

Domain-Driven Sizing

Since many of our modifications or enhancements driven by the business needs, we can size/define boundaries of our microservices that are closely aligned with Domain-Driven design & Business capabilities. But this process takes lot of time and need good domain knowledge.

Event Storming Sizing

Conducting an interactive fun session among various stake holder to identify the list of important events in the system like ‘Completed Payment’, ‘Search for a Product’ etc. Based on the events we can identify ‘Commands', 'Reactions’ and can try to group them to a domain-driven services.

Reference : https://www.lucidchart.com/blog/ddd-event-storming

RIGHT SIZING & IDENTIFYING SERVICE BOUNDARIES

eazy bytes

Now, let's take an example of a bank application that needs to be migrated or built based on a microservices architecture and attempt to determine the appropriate sizing for the services.

Saving Account & Trading Account

Cards & Loans

NOT CORRECT SIZING AS WE CAN SEE INDEPENDENT MODULES LIKE CARDS & LOANS CLUBBED TOGETHER

Saving Account

Trading Account

Cards

Loans

THIS MIGHT BE THE MOST REASONABLE CORRECT SIZING AS WE CAN SEE ALL INDEPENDENT MODULES HAVE SEPARATE SERVICE MAINTAINING LOOSELY COUPLED & HIGHLY COHESIVE

Saving Account

Trading Account

Debit Card

Credit Card

Home Loan

Vehicle Loan

Personal Loan

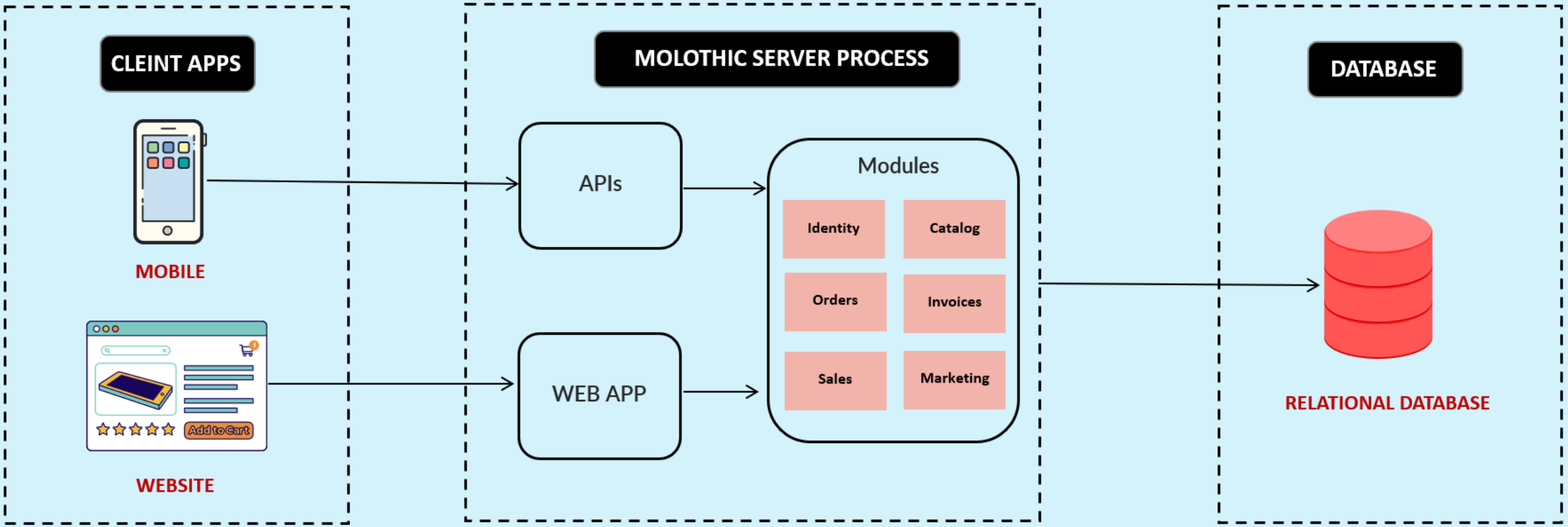
NOT CORRECT SIZING AS WE CAN SEE TOO MANY SERVICES UNDER LOANS & CARDS

MONOLOTHIC TO MICROSERVICES

MIGRATION USECASE

eazy bytes

Now let's take a scenario where an E-Commerce startup is following monolithic architecture and try to understand what's the challenges with it

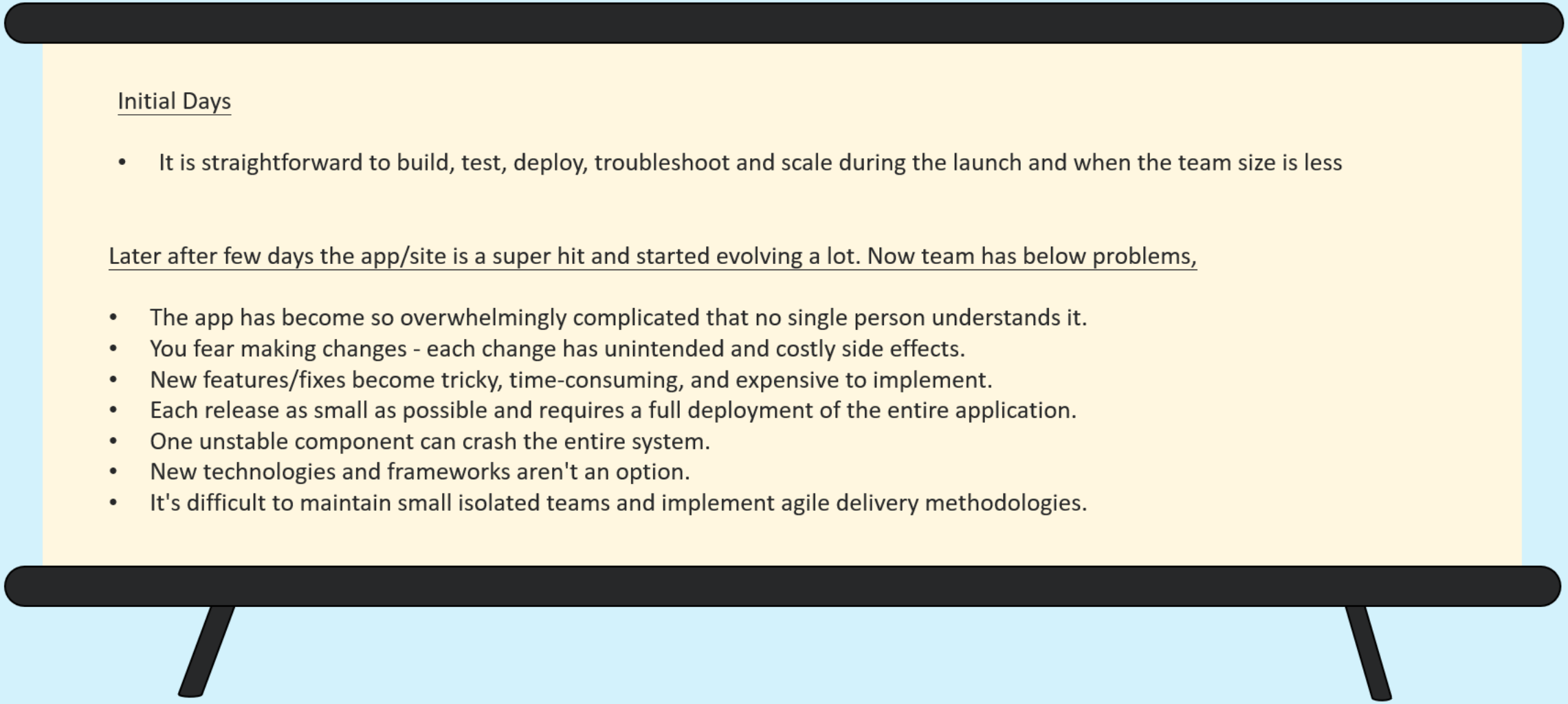


MONOLOTHIC TO MICROSERVICES

MIGRATION USECASE

eazy bytes

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



Initial Days

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

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

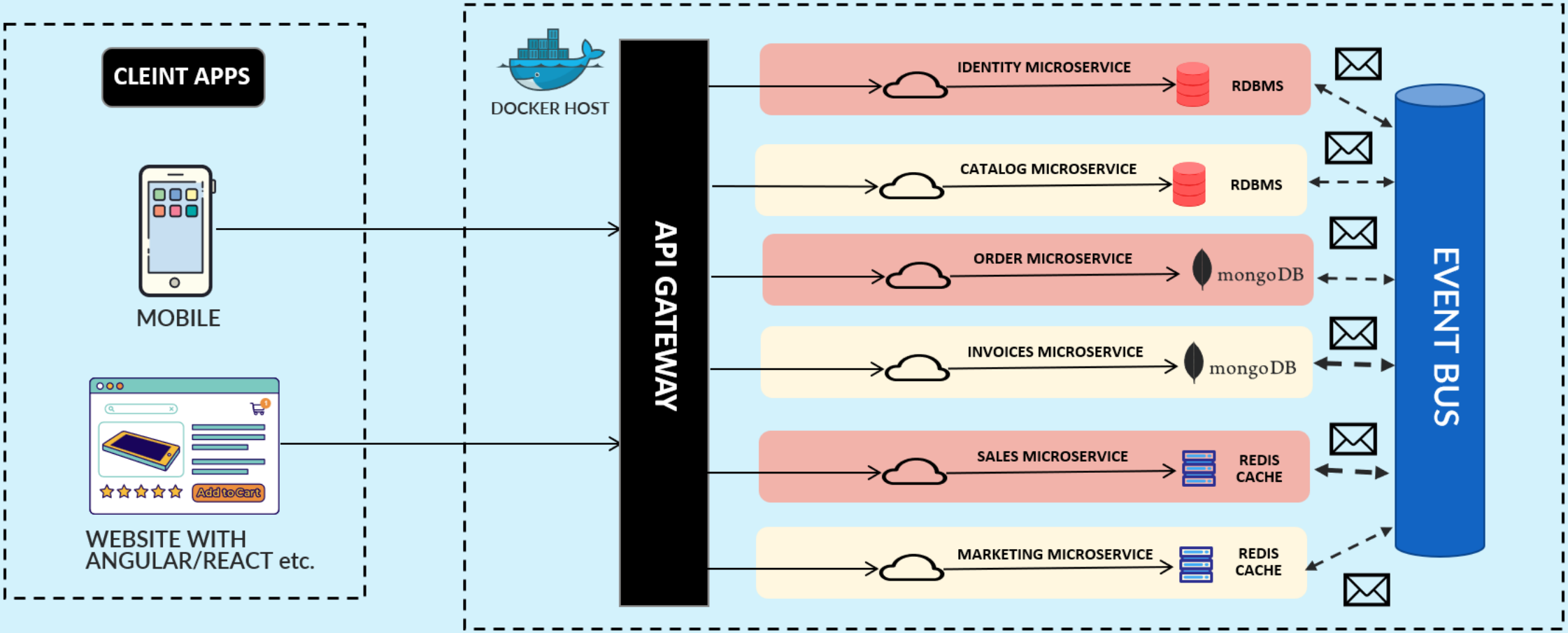
- The app has become so overwhelmingly complicated that no single person understands it.
- You fear making changes - each change has unintended and costly side effects.
- New features/fixes become tricky, time-consuming, and expensive to implement.
- Each release as small as possible and requires a full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

MONOLOTHIC TO MICROSERVICES

MIGRATION USECASE

eazy bytes

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



DEPLOYMENT, PORTABILITY & SCALABILITY OF MICROSERVICES

eazy bytes

DEPLOYEMENT

How do we deploy all the tiny 100s of microservices with less effort & cost?

PORTABILITY

How do we move our 100s of microservices across environments with less effort, configurations & cost?

SCALABILITY

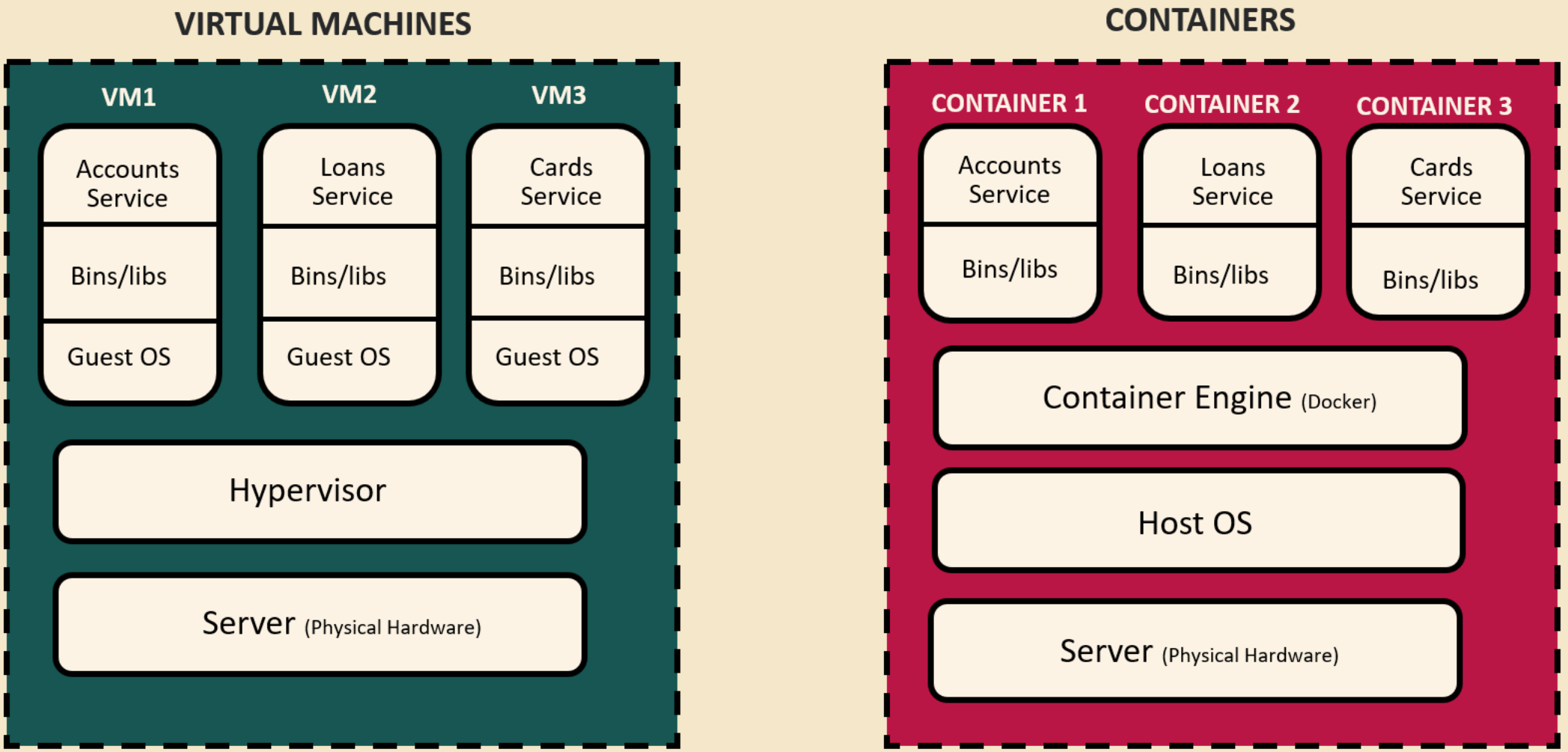
How do we scale our applications based on the demand on the fly with minimum effort & cost?

To overcome the above challenges, we should containerize our microservices. Why? Containers offer a self-contained and isolated environment for applications, including all necessary dependencies. By containerizing an application, it becomes portable and can run seamlessly in any cloud environment. Containers enable unified management of applications regardless of the language or framework used.

Docker is an open source platform that "provides the ability to package and run an application in a loosely isolated environment called a container"

WHAT ARE CONTAINERS & HOW THEY ARE DIFFERENT FROM VMs ?

eazy bytes



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

WHAT ARE CONTAINERS & Docker ?

What is a container ?

A container is a loosely isolated environment that allows us to build and run software packages. These software packages include the code and all dependencies to run applications quickly and reliably on any computing environment. We call these packages as container images.

What is software containerization ?

Software containerization is an OS virtualization method that is used to deploy and run containers without using a virtual machine (VM). Containers can run on physical hardware, in the cloud, VMs, and across multiple OSs.

What is Docker ?

Docker is an open-source platform that enables developers to automate the deployment, scaling, and management of applications using containerization. Containers are lightweight, isolated environments that encapsulate an application along with its dependencies, libraries, and runtime components.

WHAT EXACTLY HAPPENS IN CONTAINERIZATION ?

eazy bytes

Containers are based on the concept of operating system (OS) virtualization, where multiple containers can run on the same physical or virtual machine, sharing the same OS kernel. This differs from traditional virtualization, where each virtual machine (VM) runs a separate OS instance.

In containerization, Linux features such as namespaces and cgroups play a crucial role in providing isolation and resource management. Here's a brief explanation of these concepts:

NAMESPACES

Linux namespaces allow for the creation of isolated environments within the operating system. Each container has its own set of namespaces, including process, network, mount, IPC (interprocess communication), and user namespaces. These namespaces ensure that processes within a container are only aware of and can interact with resources within their specific namespace, providing a level of isolation.

CONTROL GROUPS

cgroups provide resource management and allocation capabilities for containers. They allow administrators to control and limit the resources (such as CPU, memory, disk I/O, and network bandwidth) that containers can consume. By using cgroups, container runtimes can enforce resource restrictions and prevent one container from monopolizing system resources, ensuring fair allocation among containers.

Here you may have a question. If containerization works based on linux concepts like kernel, namespaces, cgroups etc. then how is Docker supposed to work on a macOS or Windows machine. Let's try to understand the same in next slide....

HOW DOES DOCKER WORKS ON MAC & WINDOWS OS ?

eazy bytes

When you install Docker on a Linux OS, you receive the complete Docker Engine on your Linux host. However, if you opt for Docker Desktop for Mac or Windows, only the Docker client is installed on your macOS or Windows host. Behind the scenes, a lightweight virtual machine is configured with Linux, and the Docker server component is installed within that virtual machine.

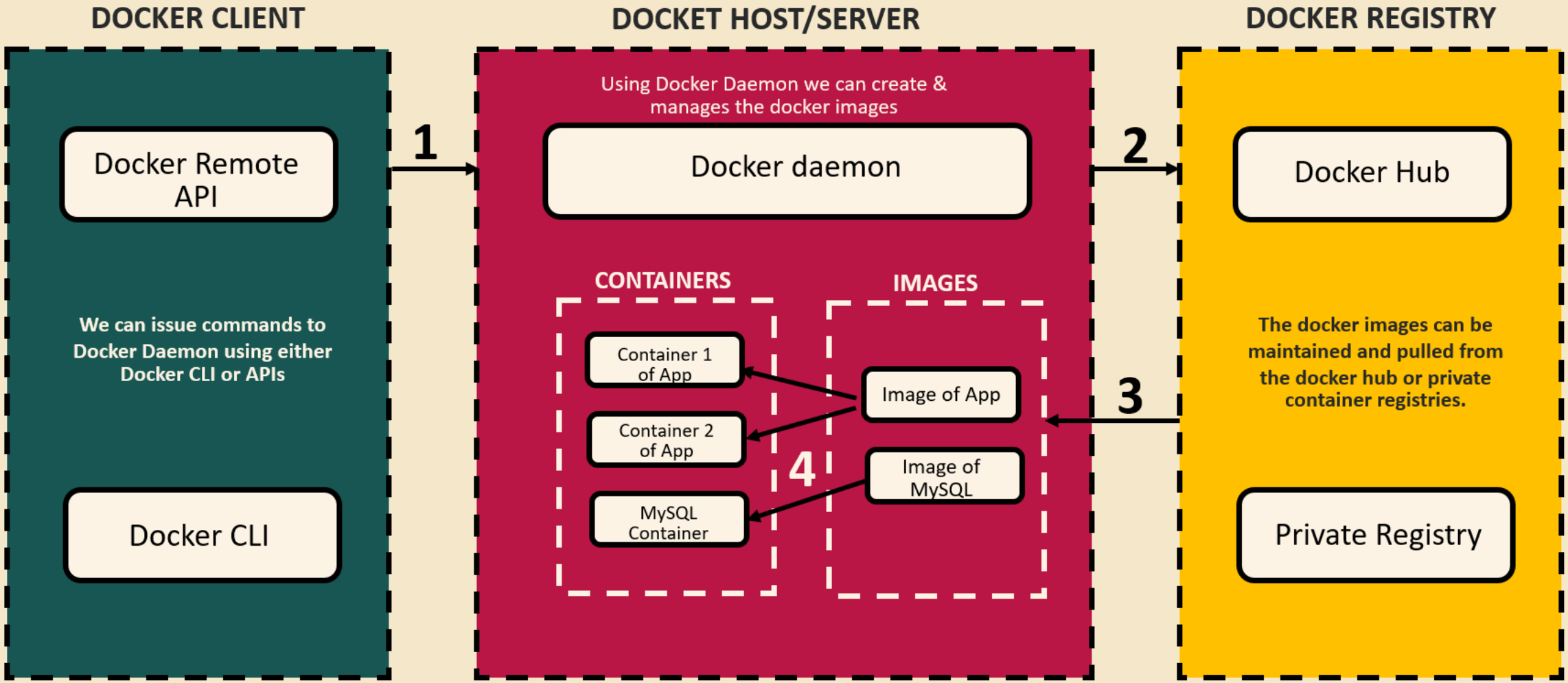
As a user, your experience will be very similar to using Docker on a Linux machine, with minimal noticeable differences. However, when you utilize the Docker CLI to execute commands, you are actually interacting with a Docker server running on a separate machine, which in this case is the Linux-based virtual machine.

To confirm this configuration, you can start Docker and execute the "docker version" command. You will observe that the Docker client is running on the darwin/amd64 architecture (on macOS) or windows/amd64 (on Windows), while the Docker server is operating on the linux/amd64 architecture.

```
eazybytes@Eazys-MBP ~ % docker version
Client:
 Cloud integration: v1.0.31
 Version:           23.0.5
 API version:       1.42
 Go version:        go1.19.8
 Git commit:        bc4487a
 Built:             Wed Apr 26 16:12:52 2023
 OS/Arch:           darwin/arm64
 Context:           default

Server: Docker Desktop 4.19.0 (106363)
 Engine:
  Version:          23.0.5
  API version:      1.42 (minimum version 1.12)
  Go version:       go1.19.8
  Git commit:       94d3ad6
  Built:            Wed Apr 26 16:17:14 2023
  OS/Arch:          linux/arm64
  Experimental:     false
 containerd:
  Version:          1.6.20
  GitCommit:        2806fc1057397dbaeefbea0e4e17bddfbd388f38
 runc:
  Version:          1.1.5
  GitCommit:        v1.1.5-0-gf19387a
 docker-init:
  Version:          0.19.0
  GitCommit:        de40ad0
```

DOCKER ARCHITECTURE ?

eazy bytes



1. Instruction from Docker Client to Server to run a container
2. Docker server finds the image in registry if not found locally
3. Docker server pulls the image from registry into local
4. Docker server creates a running container from the image

GENERATE DOCKER IMAGES

eazy bytes

To generare docker images from our existing microservices, we will explore the below three different commonly used approaches. We can choose one of them for the rest of the course

01

Dockerfile -> accounts

We need to write a dockerfile with the list of instructions which can be passed to Docker server to generate a docker image based on the given instructions

02

Buildpacks -> loans

Buildpacks (https://buildpacks.io), a project initiated by Heroku & Pivotal and now hosted by the CNCF. It simplifies containerization since with it, we don't need to write a low-level dockerfile.

03

Google Jib -> cards

Jib is an open-source Java tool maintained by Google for building Docker images of Java applications. It simplifies containerization since with it, we don't need to write a low-level dockerfile.

Running a Spring Boot app as a container using Dockerfile

eazy bytes

STEPS TO BE FOLLOWED

1) Run the maven command, "mvn clean install" from the location where pom.xml is present to generate a fat jar inside target folder

2) Write instructions to Docker inside a file with the name **Dockerfile** to generate a Docker image. Sample instructions are mentioned on the left hand side

3) Execute the docker command "docker build . -t eazybytes/accounts:s4" from the location where Dockerfile is present. This will generate the docker image based on the tag name provided

4) Execute the docker command "docker run -p 8080:8080 eazybytes/accounts:s4". This will start the docker container based on the docker image name and port mapping provided

Sample Dockerfile

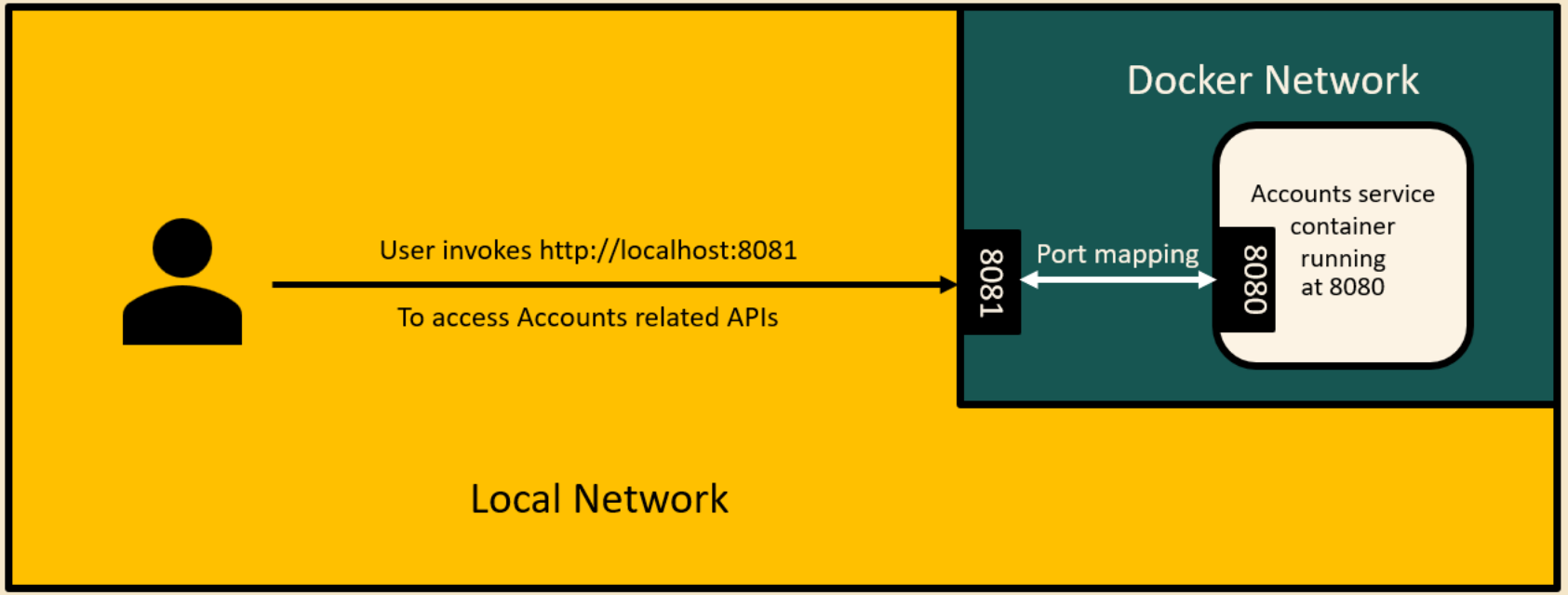
```
#Start with a base image containing Java runtime
FROM openjdk:17-jdk-slim

#Information around who maintains the image
MAINTAINER eazybytes.com

# Add the application's jar to the container
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"]
```

PORT MAPPING IN DOCKER

eazy bytes

What is port mapping or port forwarding or port publishing ?

By default, containers are connected to an isolated network within the Docker host. To access a container from your local network, you need to configure port mapping explicitly. For instance, when running the accounts Service application, we can provide the port mapping as an argument in the docker run command: **-p 8081:8080** (where the first value represents the external port and the second value represents the container port). Below diagram demonstrates the functionality of this configuration.



Running a Spring Boot app as a container using Buildpacks

eazy bytes

STEPS TO BE FOLLOWED

1) Add the configurations like mentioned on the right hand side inside the pom.xml. Make sure to pass the image name details

2) Run the maven command "mvn spring-boot:build-image" from the location where pom.xml is present to generate the docker image with out the need of Dockerfile

3) Execute the docker command "docker run -p 8090:8090 eazybytes/loans:s4". This will start the docker container based on the docker image name and port mapping provided

Sample pom.xml config

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <configuration>
        <image>
          <name>eazybytes/${project.artifactId}:s4</name>
        </image>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Cloud Native Buildpacks offer an alternative approach to Dockerfiles, prioritizing consistency, security, performance, and governance. With Buildpacks, developers can automatically generate production-ready OCI images from their application source code without the need to write a Dockerfile.

Running a Spring Boot app as a container using Google Jib

eazy bytes

STEPS TO BE FOLLOWED

1) Add the configurations like mentioned on the right hand side inside the pom.xml. Make sure to pass the image name details

2) Run the maven command "**mvn compile jib:dockerBuild**" from the location where pom.xml is present to generate the docker image with out the need of Dockerfile

3) Execute the docker command "docker run -p 9000:9000 eazybytes/cards:s4". This will start the docker container based on the docker image name and port mapping provided

Sample pom.xml config

```
<build>
  <plugins>
    <plugin>
      <groupId>com.google.cloud.tools</groupId>
      <artifactId>jib-maven-plugin</artifactId>
      <version>3.3.2</version>
      <configuration>
        <to>
          <image>eazybytes/${project.artifactId}:s4</image>
        </to>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Google Jib offer an alternative approach to Dockerfiles, prioritizing consistency, security, performance, and governance. With Jib, developers can automatically generate production-ready OCI images from their application source code without the need to write a Dockerfile and even local Docker setup.

Using Docker Compose to handle multiple containers

Why can't we run multiple containers using CLI ?

When dealing with the execution of multiple containers, utilizing the Docker CLI can be error-prone. Writing commands directly in a Terminal window can introduce errors, make the code difficult to comprehend, and pose challenges when attempting to implement version control.

What is a Docker Compose ?

It is a tool provided by Docker that allows you to define and manage multi-container applications. It uses a YAML file to describe the services, networks, and volumes required for your application. Using it, you can easily specify the configuration and relationships between different containers, making it simpler to set up and manage complex application environments.

Advantages of Docker Compose ?

By using a single command, you can create and start all the containers defined in your Docker Compose file. Docker Compose handles the orchestration and networking aspects, ensuring that the containers can communicate with each other as specified in the configuration. It also provides options for scaling services, controlling dependencies, and managing the application lifecycle.

IMPORTANT DOCKER COMMANDS

eazy bytes

01

docker images

To list all the docker images present in the Docker server

02

docker image inspect [image-id]

To display detailed image information for a given image id

03

docker image rm [image-id]

To remove one or more images for a given image ids

04

docker build . -t [image-name]

To generate a docker image based on a Dockerfile

05

docker run -p [hostport]:[containerport] [image_name]

To start a docker container based on a given image

06

docker ps

To show all running containers

07

docker ps -a

To show all containers including running and stopped

08

docker container start [container-id]

To start one or more stopped containers

09

docker container pause [container-id]

To pause all processes within one or more containers

10

docker container unpause [container-id]

To resume/unpause all processes within one or more containers

11

docker container stop [container-id]

To stop one or more running containers

12

docker container kill [container-id]

To kill one or more running containers instantly

13

docker container restart [container-id]

To restart one or more containers

14

docker container inspect [container-id]

To inspect all the details for a given container id

15

docker container logs [container-id]

To fetch the logs of a given container id

IMPORTANT DOCKER COMMANDS

eazy bytes

16 **docker container logs -f [container-id]**

To follow log output of a given container id

17 **docker rm [container-id]**

To remove one or more containers based on container ids

18 **docker container prune**

To remove all stopped containers

19 **docker image push [container_registry/ username:tag]**

To push an image from a container registry

20 **docker image pull [container_registry/ username:tag]**

To pull an image from a container registry

21 **docker image prune**

To remove all unused images

22 **docker container stats**

To show all containers statistics like CPU, memory, I/O usage

23 **Docker system prune**

Remove stopped containers, dangling images, and unused networks, volumes, and cache

24 **docker rmi [image-id]**

To remove one or more images based on image ids

25 **docker login –u [username]**

To login in to docker hub container registry

26 **docker logout**

To login out from docker hub container registry

27 **docker history [image-name]**

Displays the intermediate layers and commands that were executed when building the image

28 **docker exec -it [container-id] sh**

To open a shell inside a running container and execute commands

29 **docker compose up**

To create and start containers based on given docker compose file

30 **docker compose down**

To stop and remove containers for services defined in the Compose file

What are cloud native applications ?

eazy bytes

The layman definition

Cloud-native applications are software applications designed specifically to leverage cloud computing principles and take full advantage of cloud-native technologies and services. These applications are built and optimized to run in cloud environments, utilizing the scalability, elasticity, and flexibility offered by the cloud.

The Cloud Native Computing Foundation (CNCF) definition

Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach.

These techniques enable loosely coupled systems that are resilient, manageable, and observable. Combined with robust automation, they allow engineers to make high-impact changes frequently and predictably with minimal toil.

Important characteristics of cloud-native applications

eazy bytes

Microservices

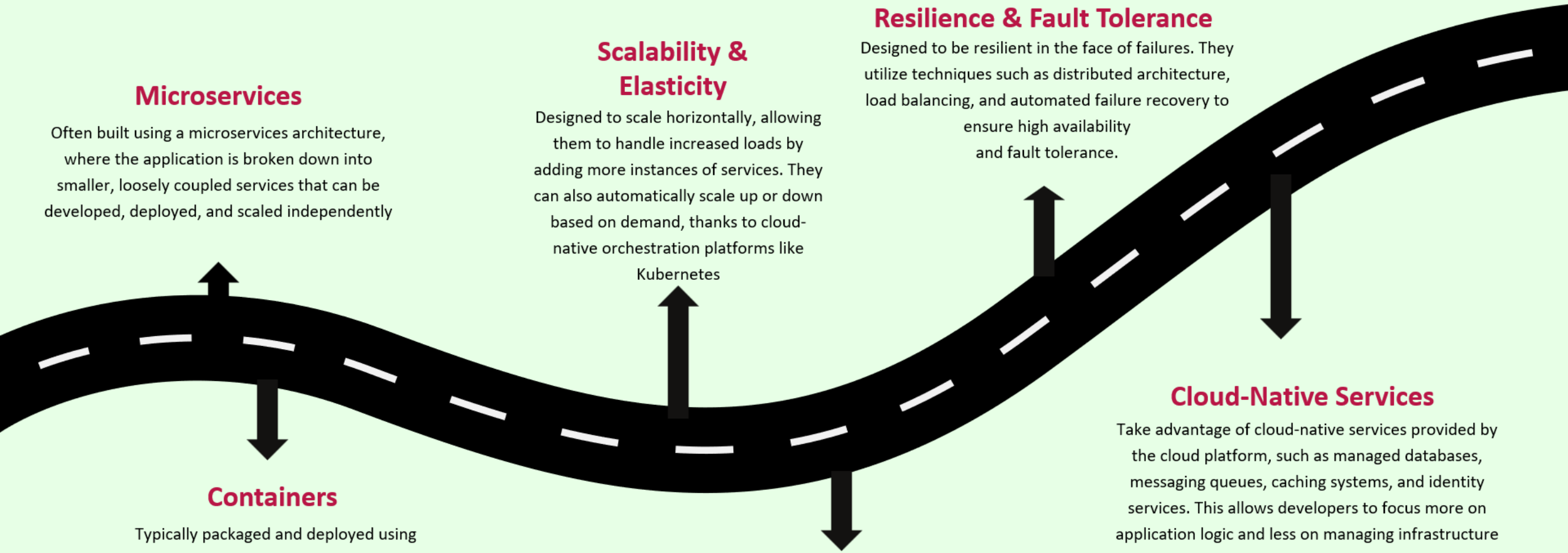
Often built using a microservices architecture, where the application is broken down into smaller, loosely coupled services that can be developed, deployed, and scaled independently

Scalability & Elasticity

Designed to scale horizontally, allowing them to handle increased loads by adding more instances of services. They can also automatically scale up or down based on demand, thanks to cloud-native orchestration platforms like Kubernetes

Resilience & Fault Tolerance

Designed to be resilient in the face of failures. They utilize techniques such as distributed architecture, load balancing, and automated failure recovery to ensure high availability and fault tolerance.



Containers

Typically packaged and deployed using containers, such as Docker containers. Containers provide a lightweight and consistent environment for running applications, making them highly portable across different cloud platforms and infrastructure

DevOps Practices

Embrace DevOps principles, promoting collaboration between development and operations teams. They often incorporate continuous integration, continuous delivery, and automated deployment pipelines to streamline the software development and deployment processes.

Cloud-Native Services

Take advantage of cloud-native services provided by the cloud platform, such as managed databases, messaging queues, caching systems, and identity services. This allows developers to focus more on application logic and less on managing infrastructure components.

DIFFERENCE B/W CLOUD-NATIVE & TRADITIONAL APPS

eazy bytes

CLOUD NATIVE APPLICATIONS	TRADITIONAL ENTERPRISE APPLICATIONS
Predictable Behavior	Unpredictable Behavior
OS abstraction	OS dependent
Right-sized capacity & Independent	Oversized capacity & Dependent
Continuous delivery	Waterfall development
Rapid recovery & Automated scalability	Slow recovery

Development Principles of Cloud Native: 12 Factors & Beyond

eazy bytes

How to get succeeded in building Cloud Native Apps & what are the guiding principles that can be considered for the same ?

The engineering team at **Heroku** cloud platform introduced the **12-Factor methodology**, a set of development principles aimed at guiding the design and construction of cloud-native applications. These principles are the result of their expertise and provide valuable insights for building web applications with specific characteristics:

1) **Cloud Platform Deployment:** Applications designed to be seamlessly deployed on various cloud platforms.
2) **Scalability as a Core Attribute:** Architectures that inherently support scalability.
3) **System Portability:** Applications that can run across different systems and environments.
4) **Enabling Continuous Deployment and Agility:** Facilitating rapid and agile development cycles.

These principles were developed to assist developers in building effective cloud-native applications, emphasizing the key factors that should be considered for optimal outcomes.

Subsequently, **Kevin Hoffman** expanded upon the original factors and introduced additional ones in his book, **"Beyond the Twelve-Factor App"** This revised approach, referred to as the **15-Factor methodology**, refreshing the content of the original principles and incorporates three new factors.

15-Factor methodology

eazy bytes

01 One codebase, one application

02 API first

03 Dependency management

04 Design, build, release, run

05 Configuration, credentials & code

06 Logs

07 Disposability

08 Backing services

09 Environment parity

10 Administrative processes

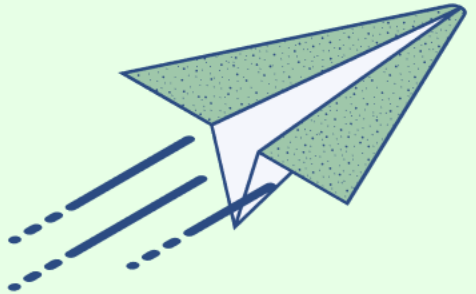
11 Port binding

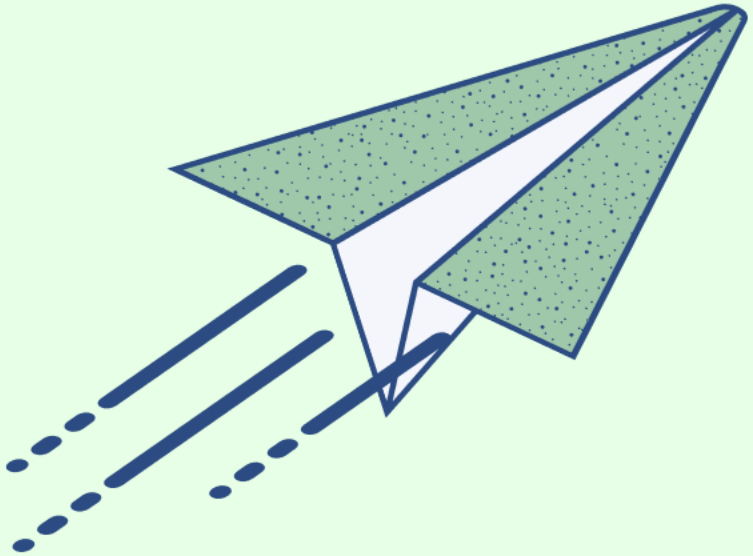
12 Stateless processes

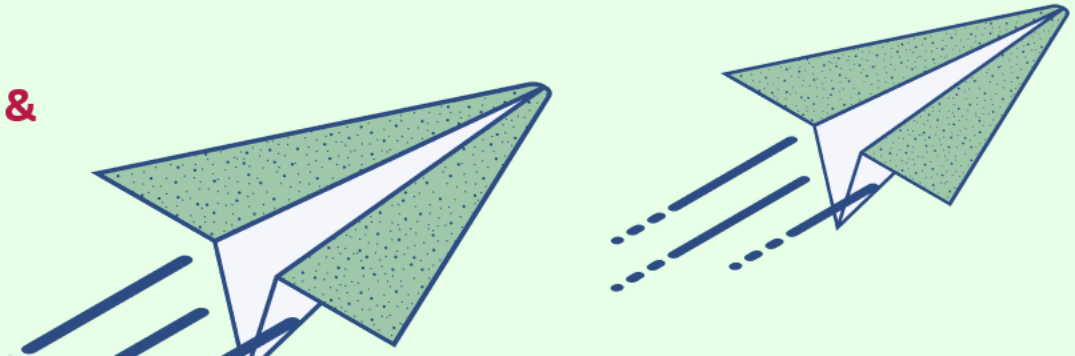
13 Concurrency

14 Telemetry

15 Authentication & authorization





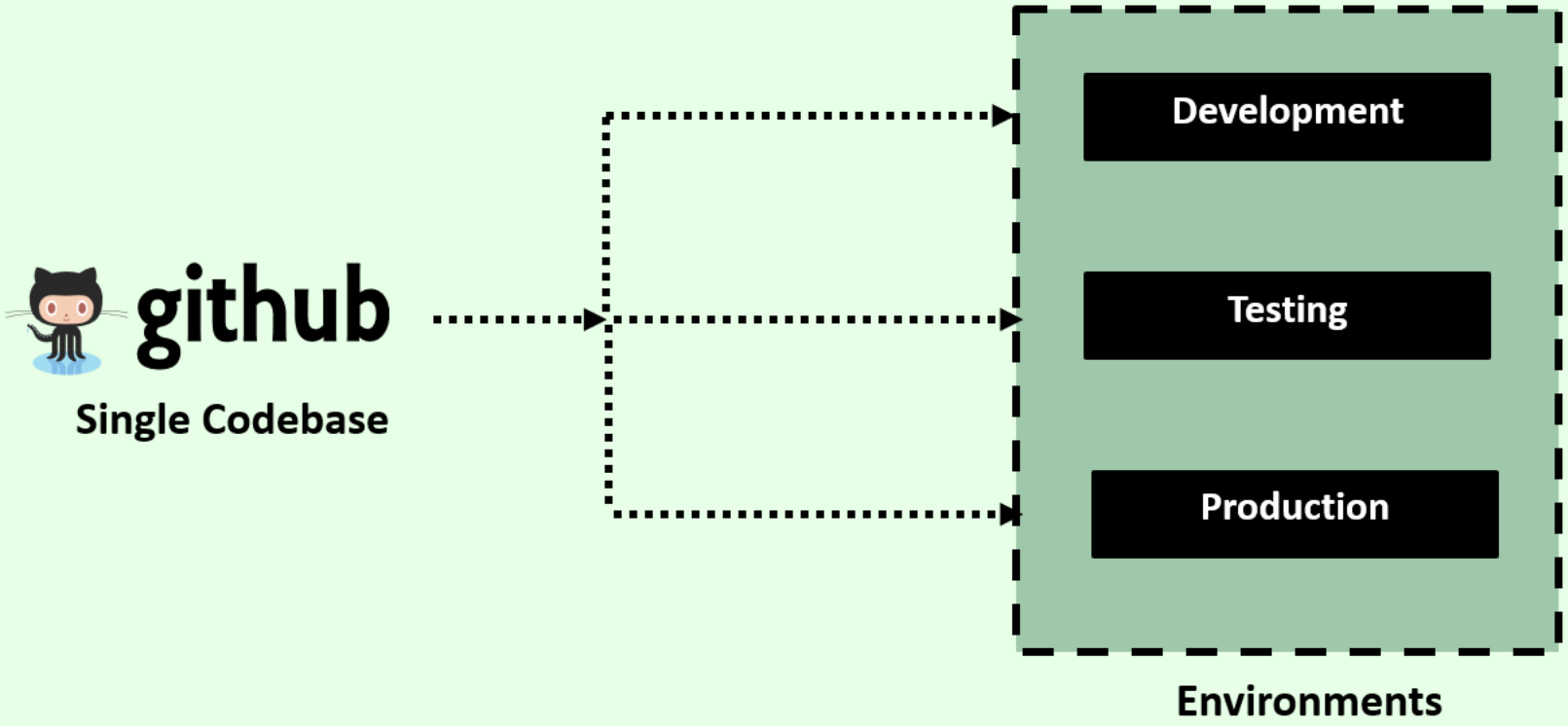


15-Factor methodology - One codebase, one application

eazy bytes

The 15-Factor methodology ensures a one-to-one correspondence between an application and its codebase, meaning each application has a dedicated codebase. Shared code is managed separately as a library, allowing it to be utilized as a dependency or as a standalone service, serving as a backing service for other applications. It is possible to track each codebase in its own repository, providing flexibility and organization.

In this methodology, a deployment refers to an operational instance of the application. Multiple deployments can exist across different environments, all leveraging the same application artifact. It is unnecessary to rebuild the codebase for each environment-specific deployment. Instead, any factors that vary between deployments, such as configuration settings, should be maintained externally from the application codebase.

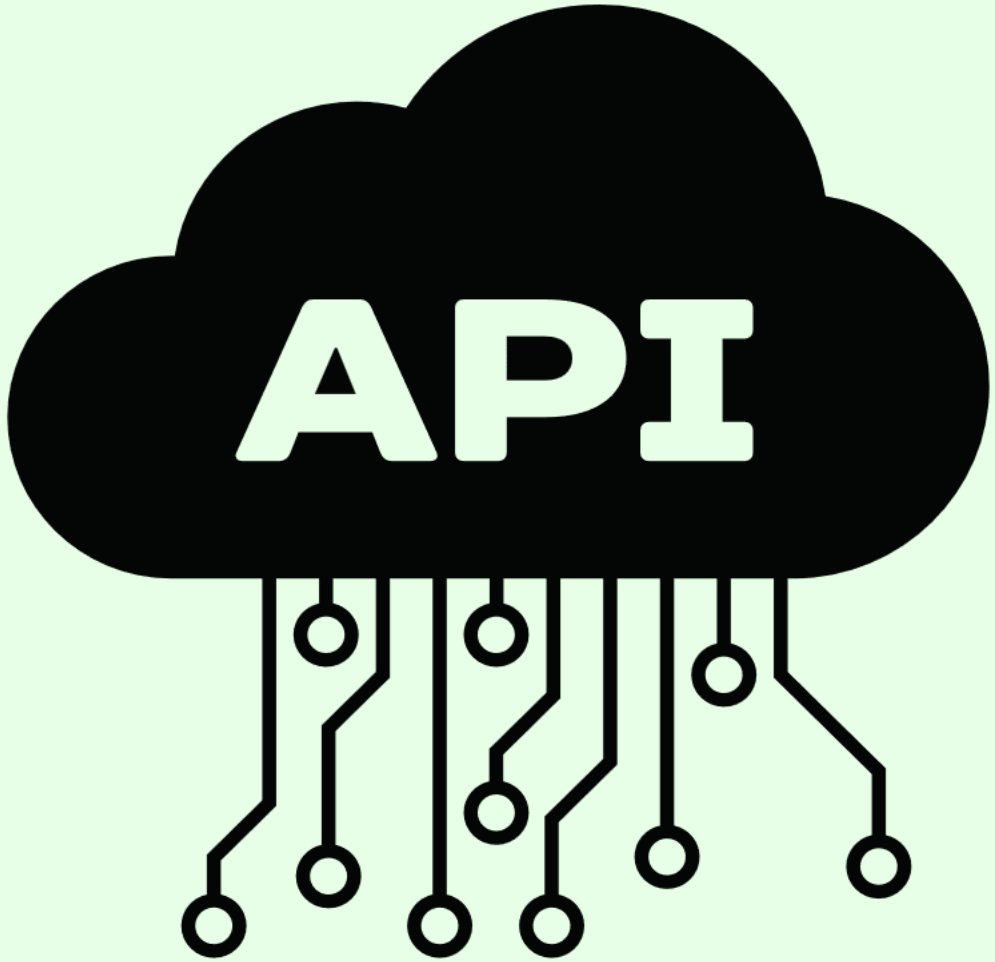


15-Factor methodology - API first

eazy bytes

In a cloud-native ecosystem, a typical setup consists of various services that interact through APIs. Adopting an API-first approach during the design phase of a cloud-native application encourages a mindset aligned with distributed systems and promotes the division of work among multiple teams. Designing the API as a priority allows other teams to build their solutions based on that API when using the application as a backing service

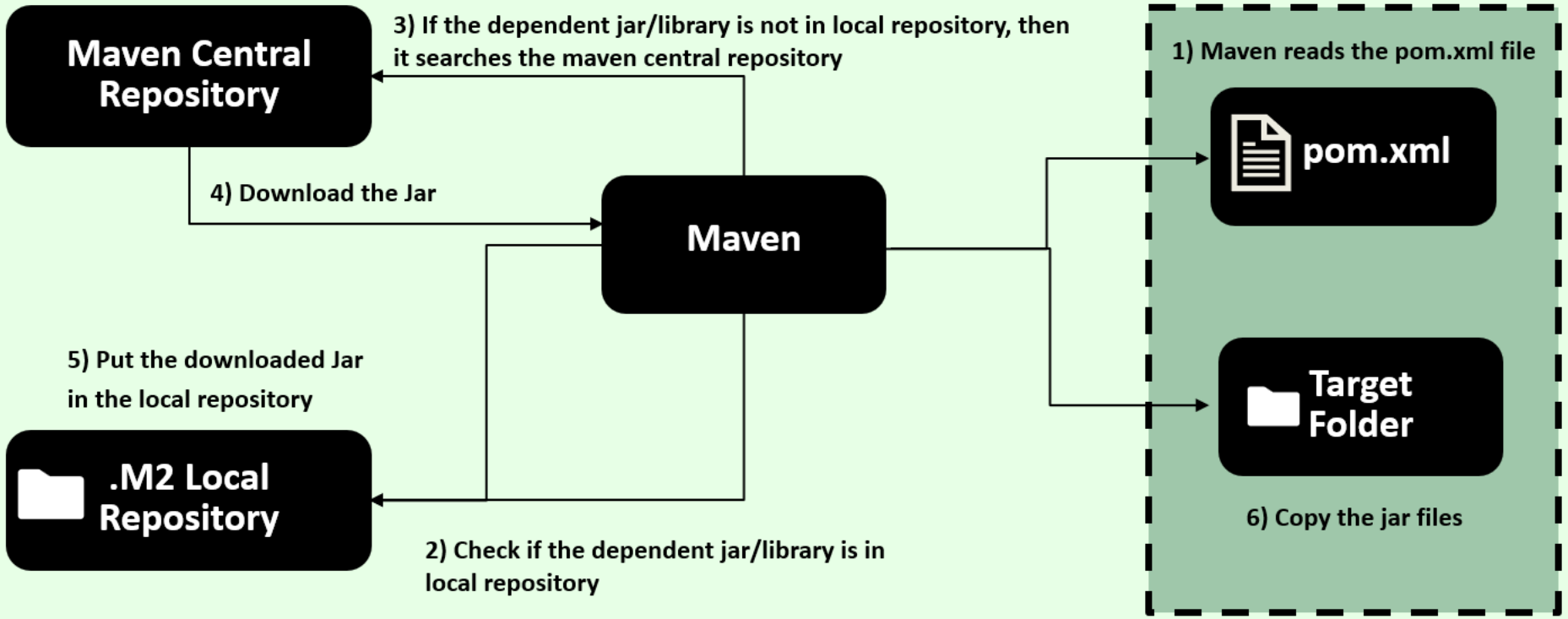
This upfront design of the API contract results in more reliable and testable integration with other systems as part of the deployment pipeline. Moreover, internal modifications to the API implementation can be made without impacting other applications or teams that rely on it



It is crucial to explicitly declare all dependencies of an application in a manifest and ensure that they are accessible to the dependency manager, which can download them from a central repository.

In the case of Java applications, we are fortunate to have robust tools like Maven or Gradle that facilitate adherence to this principle. The application should only have implicit dependencies on the language runtime and the dependency manager tool, while all private dependencies must be resolved through the dependency manager itself. By following this approach, we maintain a clear and controlled dependency management process for our application.

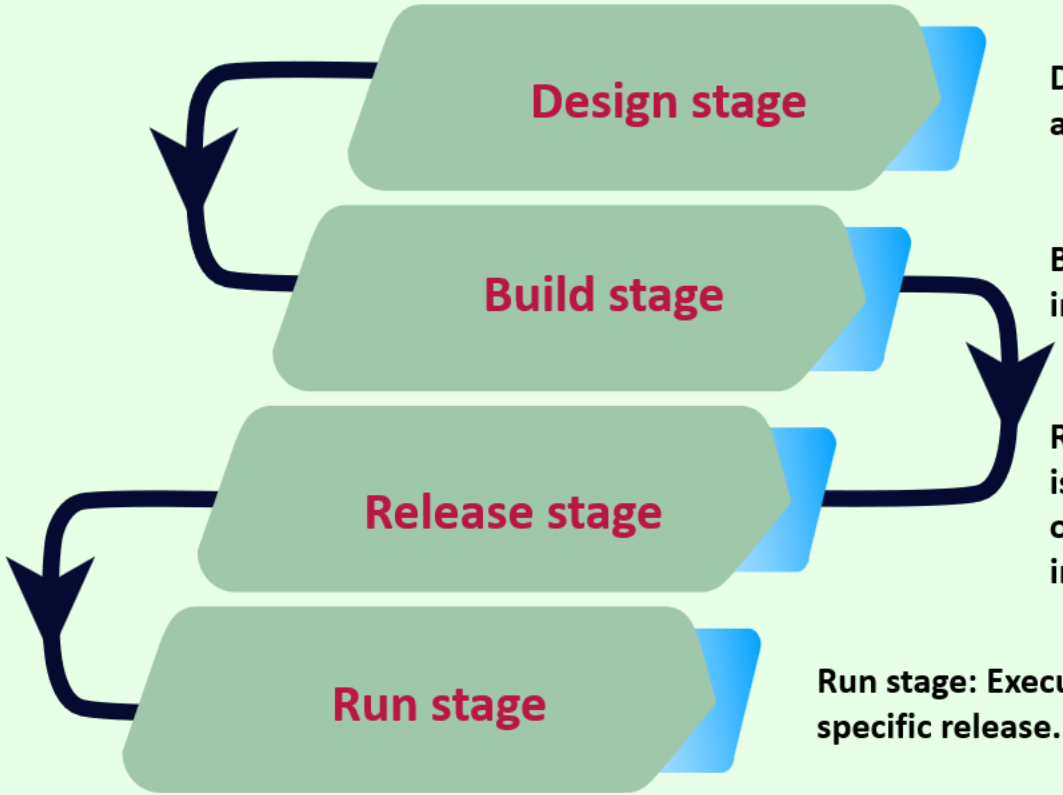
Sample flow when we use Maven as build tool



15-Factor methodology - Design, build, release, run

eazy bytes

Codebase progression from design to production deployment involves below stages,



Design stage: Determine technologies, dependencies, and tools for specific application features.

Build stage: Compile and package the codebase with dependencies, creating an immutable artifact (build). Unique identification of the build artifact is essential.

Release stage: Combine the build with a specific deployment configuration. Each release is immutable and uniquely identifiable, such as through semantic versioning (e.g., 6.1.5) or timestamp (e.g., 2023-08-15_12:01). Central repository storage facilitates easy access, including rollbacks if needed.

Run stage: Execute the application in the designated runtime environment using a specific release.

Following the 15-Factor methodology, these stages must maintain strict separation, and runtime code modifications are prohibited to prevent mismatches with the build stage. Immutable build and release artifacts should bear unique identifiers, ensuring reproducibility.

15-Factor methodology - Configuration, credentials & code

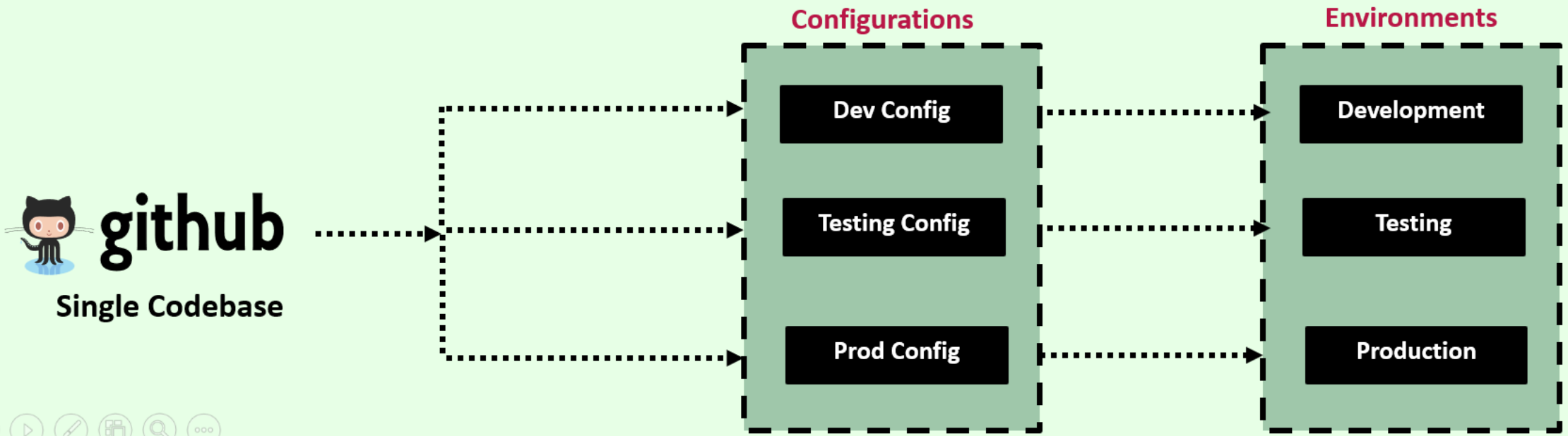
eazy bytes

According to the 15-Factor methodology, configuration encompasses all elements prone to change between deployments. It emphasizes the ability to modify application configuration independently, without code changes or the need to rebuild the application.

Configuration may include resource handles for backing services (e.g., databases, messaging systems), credentials for accessing third-party APIs, and feature flags. It is essential to evaluate whether any confidential or environment-specific information would be at risk if the codebase were exposed publicly. This assessment ensures proper externalization of configuration.

To comply with this principle, configuration should not be embedded within the code or tracked in the same codebase, except for default configuration, which can be bundled with the application. Other configurations can still be managed using separate files, but they should be stored in a distinct repository.

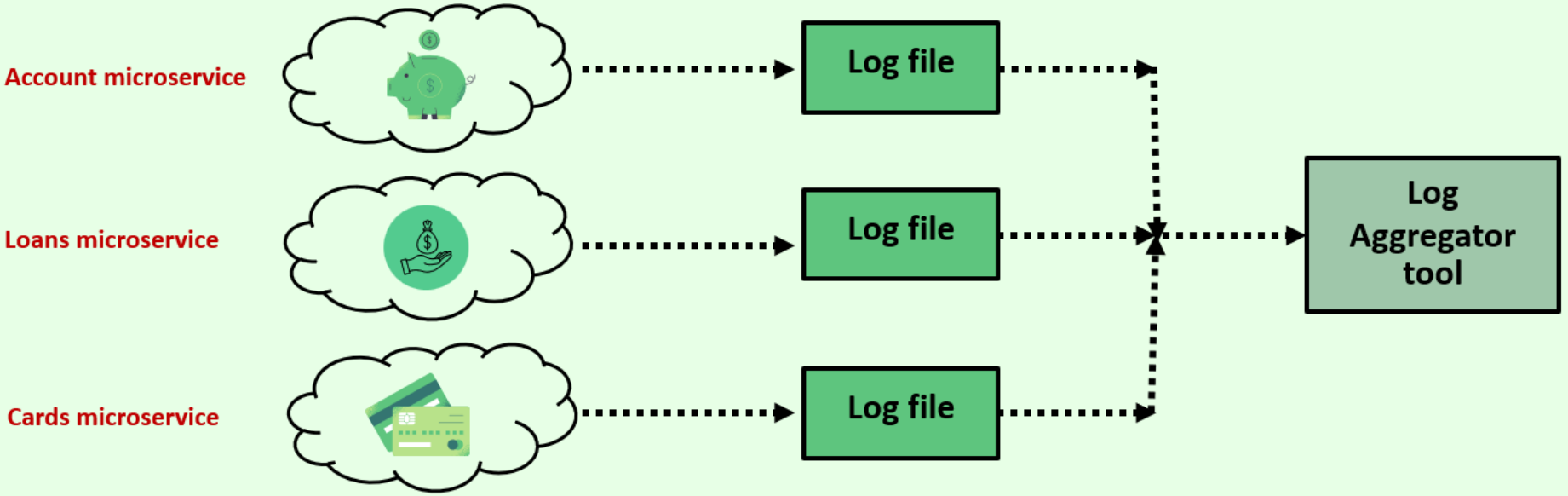
The methodology recommends utilizing environment variables to store configuration. This enables deploying the same application in different environments while adapting its behavior based on the specific environment's configuration.



15-Factor methodology - Logs

eazy bytes

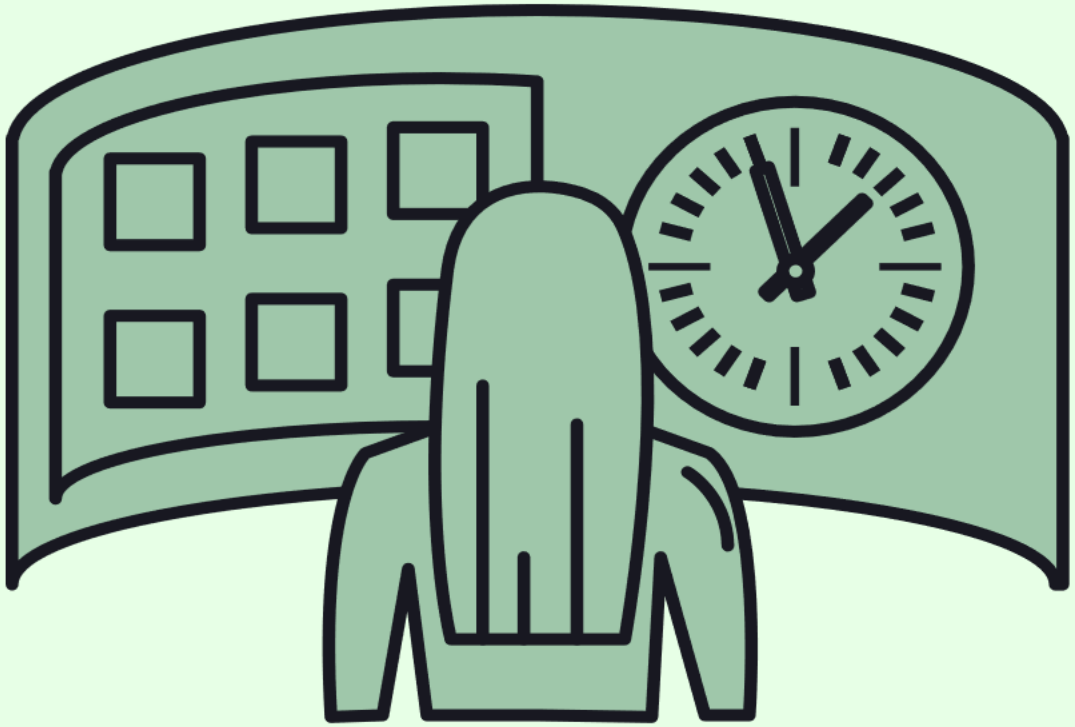
In a cloud-native application, log routing and storage are not the application's concern. Instead, applications should direct their logs to the standard output, treating them as sequentially ordered events based on time. The responsibility of log storage and rotation is now shifted to an external tool, known as a log aggregator. This tool retrieves, gathers, and provides access to the logs for inspection purposes.



15-Factor methodology - Disposability

eazy bytes

In a traditional environment, ensuring the continuous operation of applications is a top priority, striving to prevent any terminations. However, in a cloud environment, such meticulous attention is not necessary. Applications in the cloud are considered ephemeral, meaning that if a failure occurs and the application becomes unresponsive, it can be terminated and replaced with a new instance. Similarly, during high-load periods, additional instances of the application can be spun up to handle the increased workload. This concept is referred to as application disposability, where applications can be started or stopped as needed.



To effectively manage application instances in this dynamic environment, it is crucial to design them for quick startup when new instances are required and for graceful shutdown when they are no longer needed. A fast startup enables system elasticity, ensuring robustness and resilience. Without fast startup capabilities, performance and availability issues may arise.

A graceful shutdown involves the application, upon receiving a termination signal, ceasing to accept new requests, completing any ongoing ones, and then exiting. This process is straightforward for web processes. However, for worker processes or other types, it involves returning any pending jobs to the work queue before exiting.

Docker containers along with an orchestrator like Kubernetes inherently satisfy this requirement.

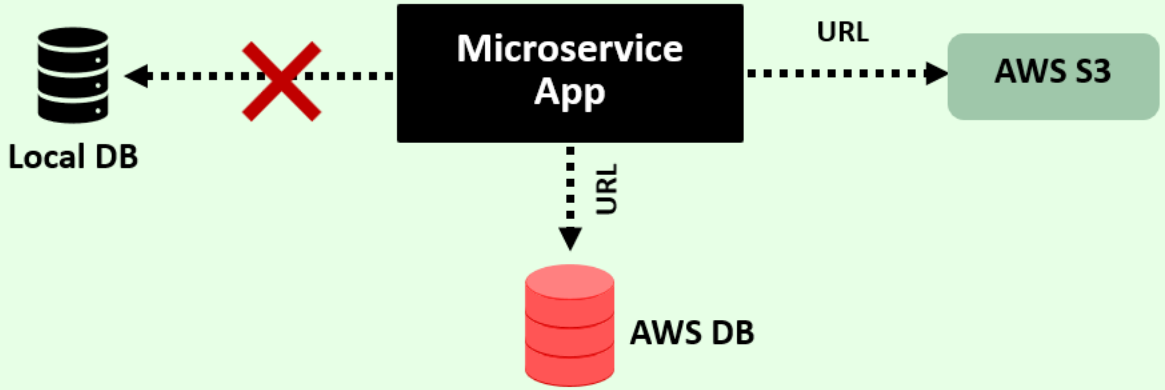
15-Factor methodology - Backing services

eazy bytes

Backing services refer to external resources that an application relies on to provide its functionality. These resources can include databases, message brokers, caching systems, SMTP servers, FTP servers, or RESTful web services. By treating these services as attached resources, you can modify or replace them without needing to make changes to the application code.

Consider the usage of databases throughout the software development life cycle. Typically, different databases are used in different stages such as development, testing, and production. By treating the database as an attached resource, you can easily switch to a different service depending on the environment. This attachment is achieved through resource binding, which involves providing necessary information like a URL, username, and password for connecting to the database.

In the below example, we can see that a local DB can be swapped easily to a third-party DB like AWS DB with out any code changes,



15-Factor methodology - Environment parity

eazy bytes

Environment parity aims to minimize differences between various environments & avoiding costly shortcuts. Here, the adoption of containers can greatly contribute by promoting the same execution environment.

There are three gaps that this factor addresses:

Time gap: The time it takes for a code change to be deployed can be significant. The methodology encourages automation and continuous deployment to reduce the time between code development and production deployment.

People gap: Developers create applications, while operators handle their deployment in production. To bridge this gap, a DevOps culture promotes collaboration between developers and operators, fostering the "you build it, you run it" philosophy.

Tools gap: Handling of backing services differs across environments. For instance, developers might use the H2 database locally but PostgreSQL in production. To achieve environment parity, it is recommended to use the same type and version of backing services across all environments.

15-Factor methodology - Administrative processes

eazy bytes

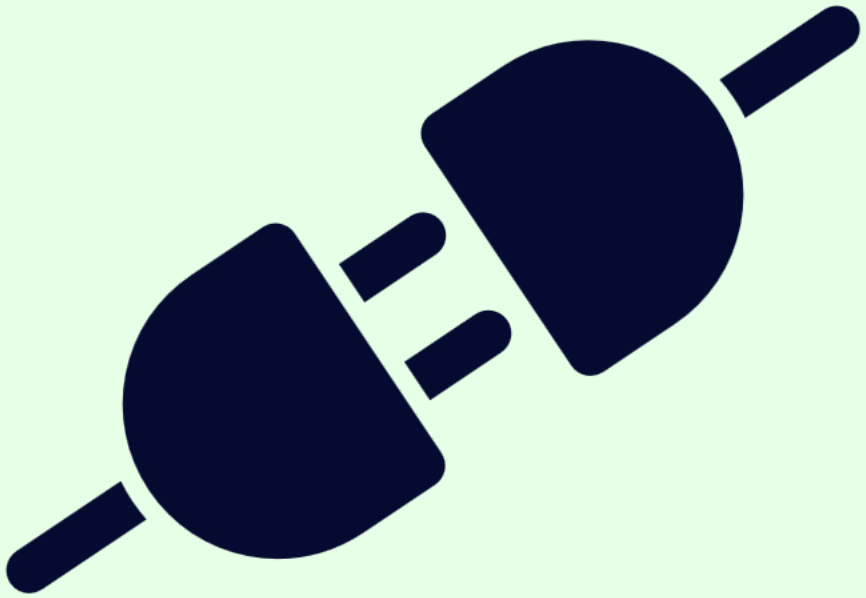
Management tasks required to support applications, such as database migrations, batch jobs, or maintenance tasks, should be treated as isolated processes. Similar to application processes, the code for these administrative tasks should be version controlled, packaged alongside the application, and executed within the same environment.

It is advisable to consider administrative tasks as independent microservices that are executed once and then discarded, or as functions configured within a stateless platform to respond to specific events. Alternatively, they can be integrated directly into the application, activated by calling a designated endpoint.

15-Factor methodology - Port binding

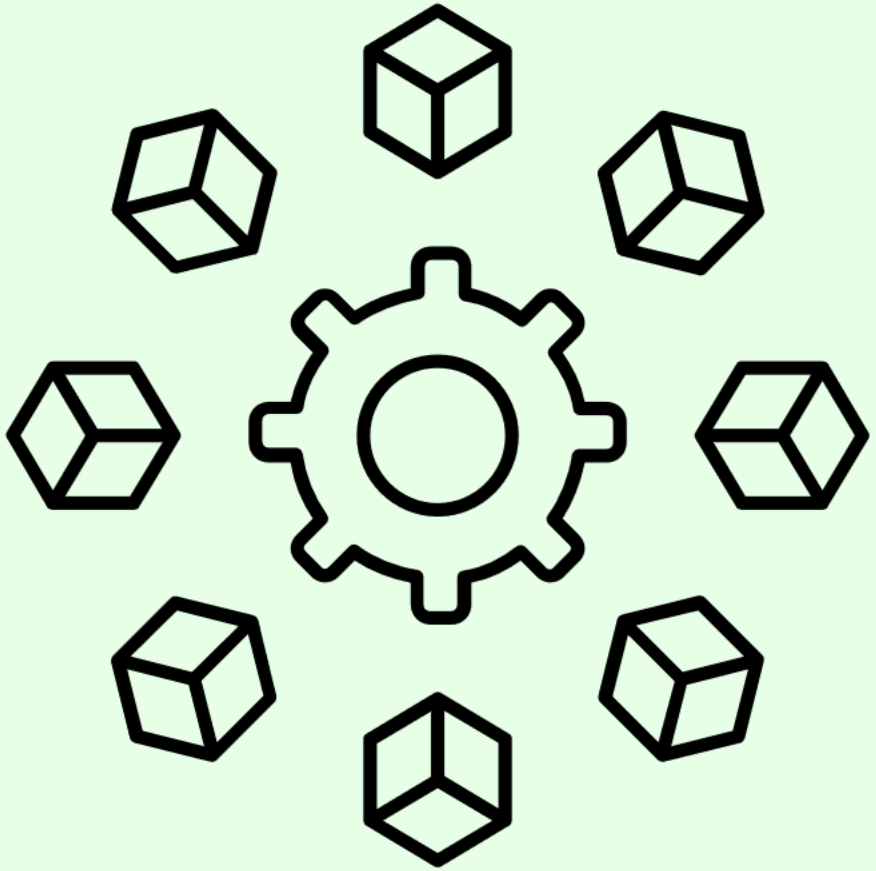
eazy bytes

Cloud native applications, adhering to the 15-Factor methodology, should be self-contained and expose their services through port binding. In production environments, routing services may be employed to translate requests from public endpoints to the internally port-bound services.



An application is considered self-contained when it doesn't rely on an external server within the execution environment. For instance, a Java web application might typically run within a server container like Tomcat, Jetty, or Undertow. In contrast, a cloud native application does not depend on the presence of a Tomcat server in the environment; it manages the server as a dependency within itself. For example, Spring Boot enables the usage of an embedded server, where the application incorporates the server instead of relying on its availability in the execution environment. Consequently, each application is mapped to its own server, diverging from the traditional approach of deploying multiple applications on a single server.

The services offered by the application are then exposed through port binding. For instance, a web application binds its HTTP services to a specific port and can potentially serve as a backing service for another application. This is a common practice within cloud native systems.



Cloud native applications are often developed with high scalability in mind. One of the key principles to achieve scalability is designing applications as stateless processes and adopting a share-nothing architecture. This means that no state should be shared among different instances of the application. It is important to evaluate whether any data would be lost if an instance of the application is destroyed and recreated. If data loss would occur, then the application is not truly stateless.

However, it's important to note that some form of state management is necessary for applications to be functional. To address this, we design applications to be stateless and delegate the handling and storage of state to specific stateful services, such as data stores. In other words, a stateless application relies on a separate backing service to manage and store the required state, while the application itself remains stateless. This approach allows for better scalability and flexibility while ensuring that necessary state is still maintained and accessible when needed.

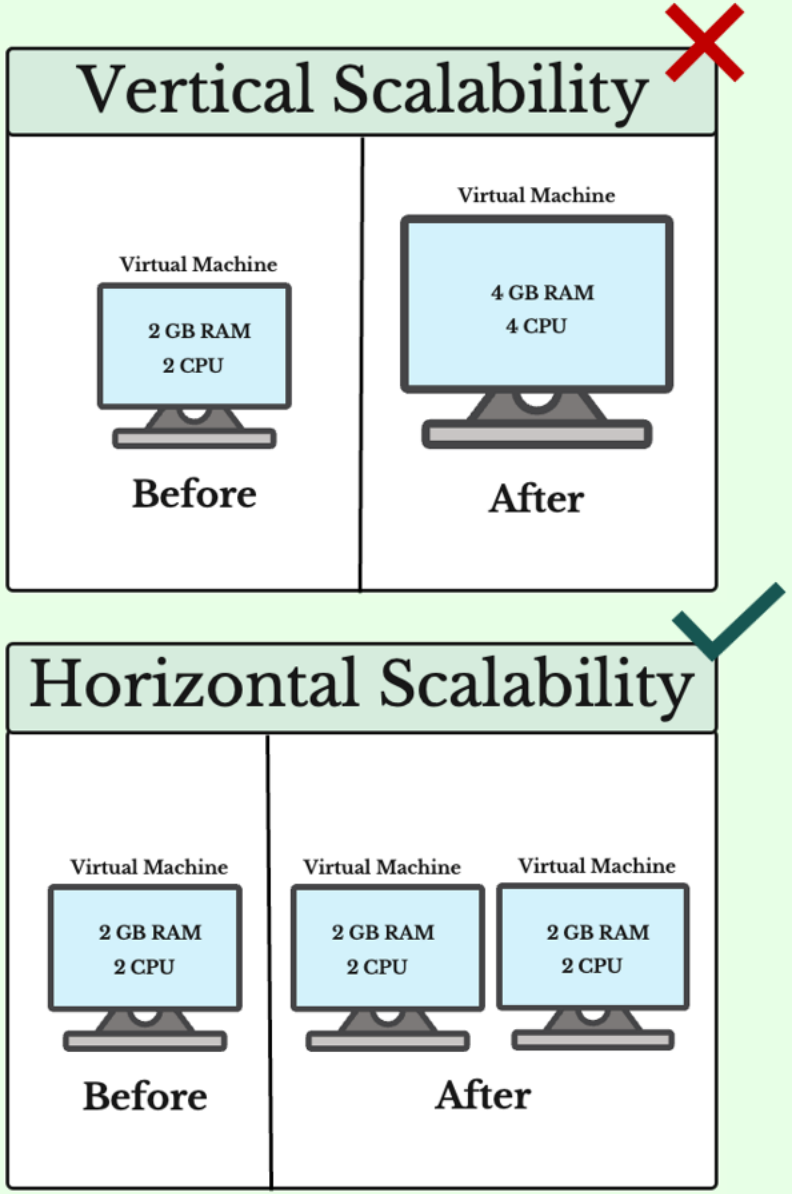
15-Factor methodology - Concurrency

eazy bytes

Scalability is not solely achieved by creating stateless applications. While statelessness is important, scalability also requires the ability to serve a larger number of users. This means that applications should support concurrent processing to handle multiple users simultaneously.

According to the 15-Factor methodology, processes play a crucial role in application design. These processes should be horizontally scalable, distributing the workload across multiple processes on different machines. This concurrency is only feasible when applications are stateless. In Java Virtual Machine (JVM) applications, concurrency is typically managed through the use of multiple threads, which are available from thread pools.

Processes can be categorized based on their respective types. For instance, there are web processes responsible for handling HTTP requests, as well as worker processes that execute scheduled background jobs. By classifying processes and optimizing their concurrency, applications can effectively scale and handle increased workloads.



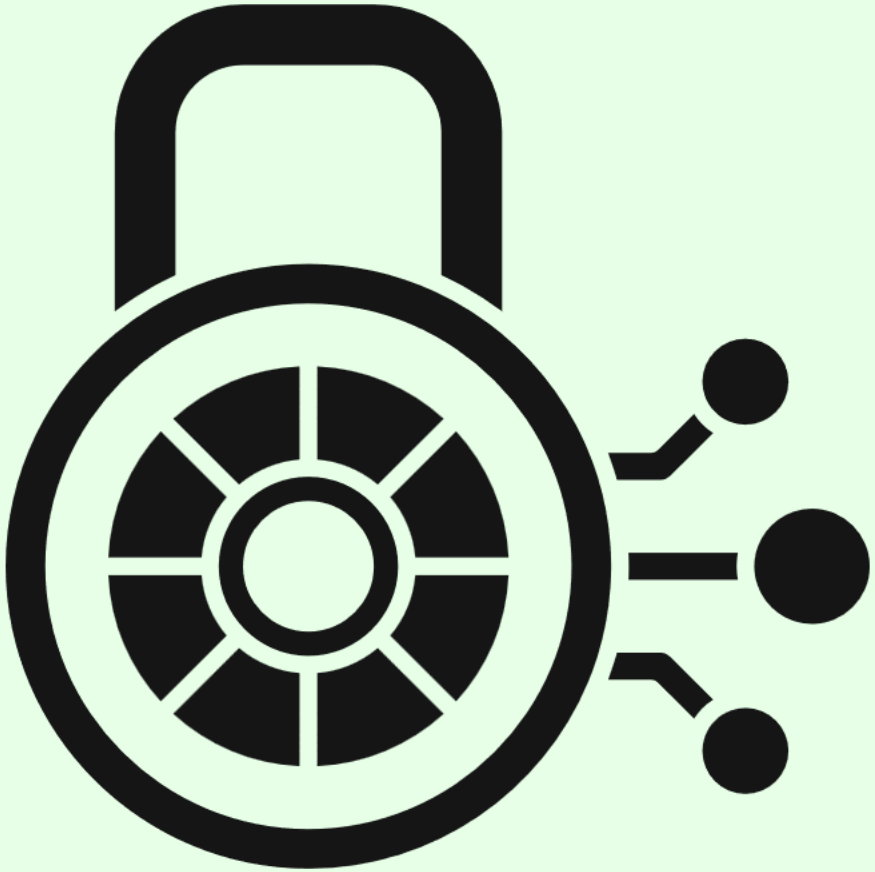
15-Factor methodology - Telemetry

eazy bytes

Observability is a fundamental characteristic of cloud native applications. With the inherent complexity of managing a distributed system in the cloud, it becomes essential to have access to accurate and comprehensive data from each component of the system. This data enables remote monitoring of the system's behavior and facilitates effective management of its intricacies. Telemetry data, such as logs, metrics, traces, health status, and events, plays a vital role in providing this visibility.

In Kevin Hoffman's analogy, he emphasizes the significance of telemetry by comparing applications to space probes. Just like telemetry is crucial for monitoring and controlling space probes remotely, the same concept applies to applications. To effectively monitor and control applications remotely, you need various types of telemetry data.

Consider the kind of telemetry that would be necessary to ensure remote monitoring and control of your applications. This includes information such as detailed logs for troubleshooting, metrics to measure performance, traces to understand request flows, health status to assess system well-being, and events to capture significant occurrences. By gathering and utilizing these types of telemetry data, you can gain valuable insights into your applications and make informed decisions to manage them effectively from a remote location.

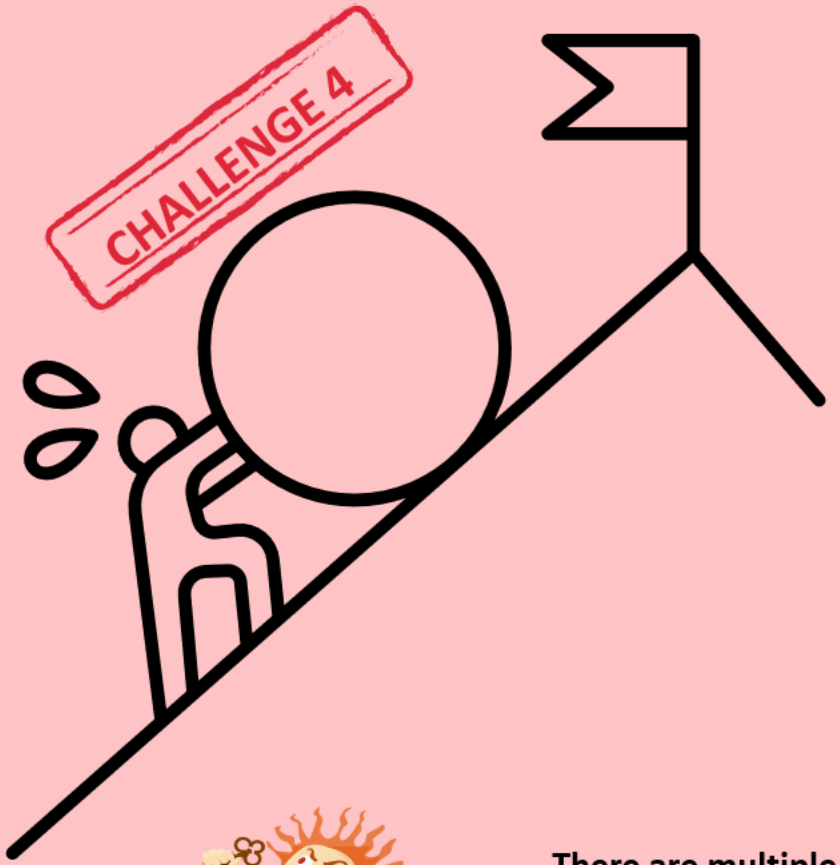


Security is a critical aspect of a software system, yet it often doesn't receive the necessary emphasis it deserves. To uphold a zero-trust approach, it is essential to ensure the security of every interaction within the system, encompassing architectural and infrastructural levels. While security involves more than just authentication and authorization, these aspects serve as a solid starting point.

Authentication enables us to track and verify the identity of users accessing the application. By authenticating users, we can then proceed to evaluate their permissions and determine if they have the necessary authorization to perform specific actions. Implementing identity and access management standards can greatly enhance security. Notable examples include OAuth 2.1 and OpenID Connect, which we will explore in this course.

CONFIGURATION MANAGEMENT IN MICROSERVICES

eazy bytes



SEPARATION OF CONFIGs/PROPERTIES

How do we separate the configurations/properties from the microservices so that same Docker image can be deployed in multiple envs.

INJECT CONFIGs/PROPERTIES

How do we inject configurations/properties that microservice needed during start up of the service

MAINTAIN CONFIGs/PROPERTIES

How do we maintain configurations/properties in a centralized repository along with versioning of them

There are multiple solutions available in Spring Boot ecosystem to handle this challenge. Below are the solutions. Let's try to identify which one suites for microservices

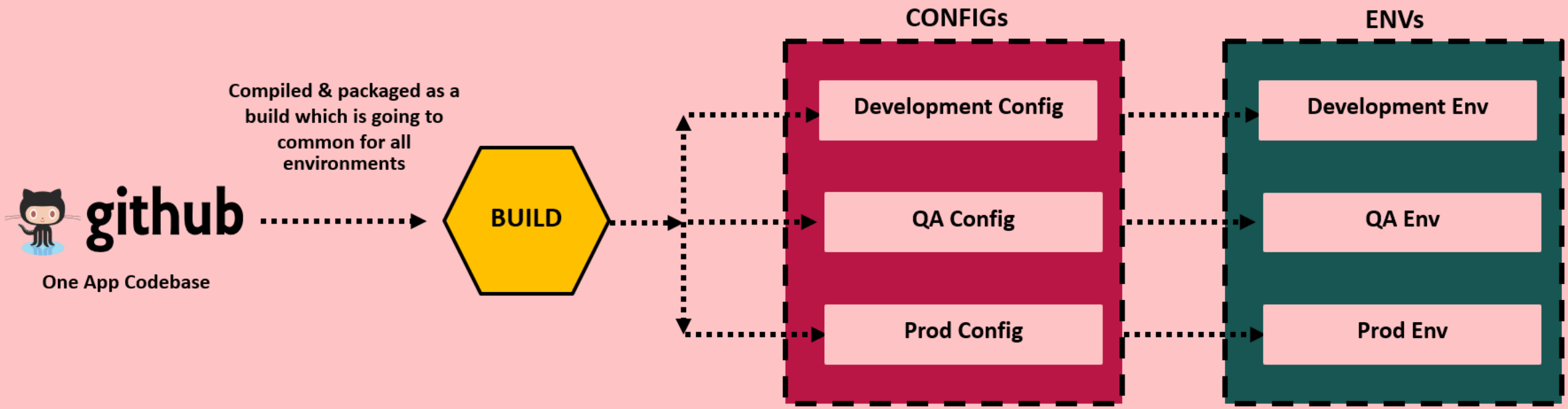
1) Configuring Spring Boot with properties and profiles
2) Applying external configuration with Spring Boot
3) Implementing a configuration server with Spring Cloud Config Server

HOW CONFIGURATIONs HANDLED IN TRADITIONAL APPs & MICROSERVICES

eazy bytes

Traditional applications were typically bundled together with their source code and various configuration files that contained environment-specific data. This meant that updating the configuration required rebuilding the entire application, or creating separate builds for each environment. As a result, there was no guarantee that the application would behave consistently across different environments, leading to potential issues when moving from staging to production.

According to the 15-Factor methodology, configuration encompasses any element likely to change between deployments, such as credentials, resource handles, and service URLs. Cloud native applications address this challenge by maintaining the immutability of the application artifact across environments. Regardless of the deployment environment, the application build remains unchanged. In cloud native applications, each deployment involves combining the build with specific configuration data. This allows the same build to be deployed to multiple environments while accommodating different configuration requirements, as shown below,



HOW CONFIGURATIONS WORK IN SPRINGBOOT ?

Spring Boot lets you externalize your configuration so that you can work with the same application code in different environments. You can use a variety of external configuration sources, include Java properties files, YAML files, environment variables, and command-line arguments.

- ✓ By default, Spring Boot look for the configurations or properties inside application.properties/yaml present in the classpath location. But we can have other property files as well and make SpringBoot to read from them.

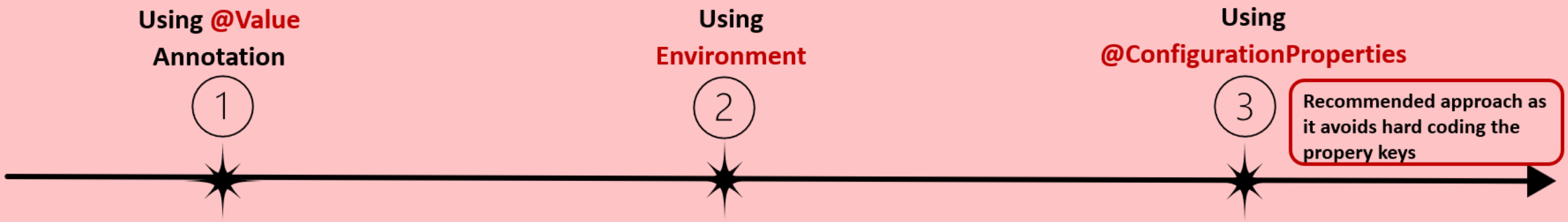
- ✓ Spring Boot uses a very particular order that is designed to allow sensible overriding of values. Properties are considered in the following order (with values from lower items overriding earlier ones):

- Properties present inside files like application.properties
- OS Environmental variables
- Java System properties (System.getProperties())
- JNDI attributes from java:comp/env
- ServletContext init parameters
- ServletConfig init parameters
- Command line arguments

HOW TO READ PROPERTIES IN SPRINGBOOT APPs

eazy bytes

In Spring Boot, there are multiple approaches to reading properties. Below are the most commonly used approaches,



You can use the @Value annotation to directly inject property values into your beans. This approach is suitable for injecting individual properties into specific fields.For example:

```
@Value("${property.name}")
private String propertyValue;
```

The Environment interface provides methods to access properties from the application's environment. You can autowire the Environment bean and use its methods to retrieve property values. This approach is more flexible and allows accessing properties programmatically. For example:

```
@Autowired
private Environment environment;

public void getProperty() {
   String propertyValue =
      environment.getProperty("property.name");
}
```

The @ConfigurationProperties annotation enables binding of entire groups of properties to a bean. You define a configuration class with annotated fields matching the properties, and Spring Boot automatically maps the properties to the corresponding fields.

```
@ConfigurationProperties("prefix")
public class MyConfig {
   private String property;

   // getters and setters
}
```

In this case, properties with the prefix "prefix" will be mapped to the fields of the MyConfig class.

Profiles

Spring provides a great tool for grouping configuration properties into so-called profiles(dev, qa, prod) allowing us to activate a bunch of configurations based on the active profile.

Profiles are perfect for setting up our application for different environments, but they're also being used in another use cases like Bean creation based on a profile etc.

So basically a profile can influence the application properties loaded and beans which are loaded into the Spring context.

eazy bytes

The default profile is always active. Spring Boot loads all properties in application.properties into the default profile.

We can create another profiles by creating property files like below,

application_prod.properties ---------> for prod profile
application_qa.properties ---------> for QA profile

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

spring.profiles.active=prod

An important point to consider is that once an application is built and packaged, it should not be modified. If any configuration changes are required, such as updating credentials or database handles, they should be made externally.

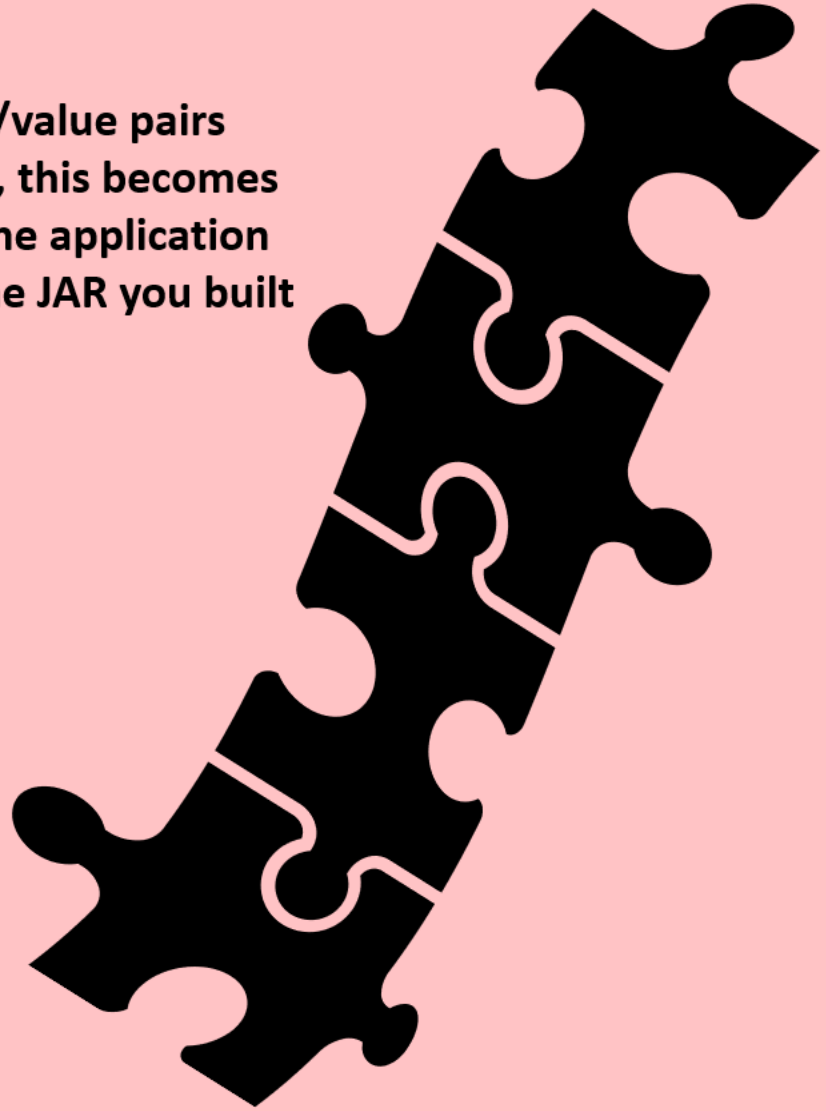
How to externalize configurations using command-line arguments ?

eazy bytes

Spring Boot 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. You can customize the application configuration by specifying command-line arguments when running the JAR you built earlier.

```
java -jar accounts-service-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 arguments.



How to externalized configurations using JVM system properties ?

eazy bytes

JVM system properties, similar to command-line arguments, can override Spring properties with a lower priority. This approach allows for externalizing the configuration without the need to rebuild the JAR artifact. The JVM system property follows the same naming convention as the corresponding Spring property, prefixed with -D for JVM arguments. In the application, the message defined as a JVM system property will be utilized, taking precedence over property files.

```
java -Dbuild.version=”1.2" -jar accounts-service-0.0.1-SNAPSHOT.jar
```

In the scenario where both a JVM system property and a command-line argument are specified, the precedence rules dictate that Spring will prioritize the value provided as a command-line argument. This means that the value specified through the CLI will be utilized by the application, taking precedence over the JVM properties.

How to externalized configurations using environment variables ?

eazy bytes

Environment variables are widely used for externalized configuration as they offer portability across different operating systems, as they are universally supported. Most programming languages, including Java, provide mechanisms to access environment variables, such as the System.getenv() method.

To map a Spring property key to an environment variable, you need to convert all letters to uppercase and replace any dots or dashes with underscores. Spring Boot will handle this mapping correctly internally. For example, an environment variable named BUILD_VERSION will be recognized as the property build.version. This feature is known as relaxed binding.

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
```

Drawbacks of externalized configurations using SpringBoot alone

eazy bytes

1. CLI arguments, JVM properties, and environment variables are effective ways to externalize configuration and maintain the immutability of the application build. However, using these approaches often involves executing separate commands and manually setting up the application, which can introduce potential errors during deployment.

2. Given that configuration data evolves and requires changes, similar to application code, what strategies should be employed to store, track revisions and audit the configuration used in a release?

3. In scenarios where environment variables lack granular access control features, how can you effectively control access to configuration data?

4. When the number of application instances grows, handling configuration in a distributed manner for each instance becomes challenging. How can such challenges be overcome?

5. Considering that neither Spring Boot properties nor environment variables support configuration encryption, how should secrets be managed securely?

6. After modifying configuration data, how can you ensure that the application can read it at runtime without necessitating a complete restart?

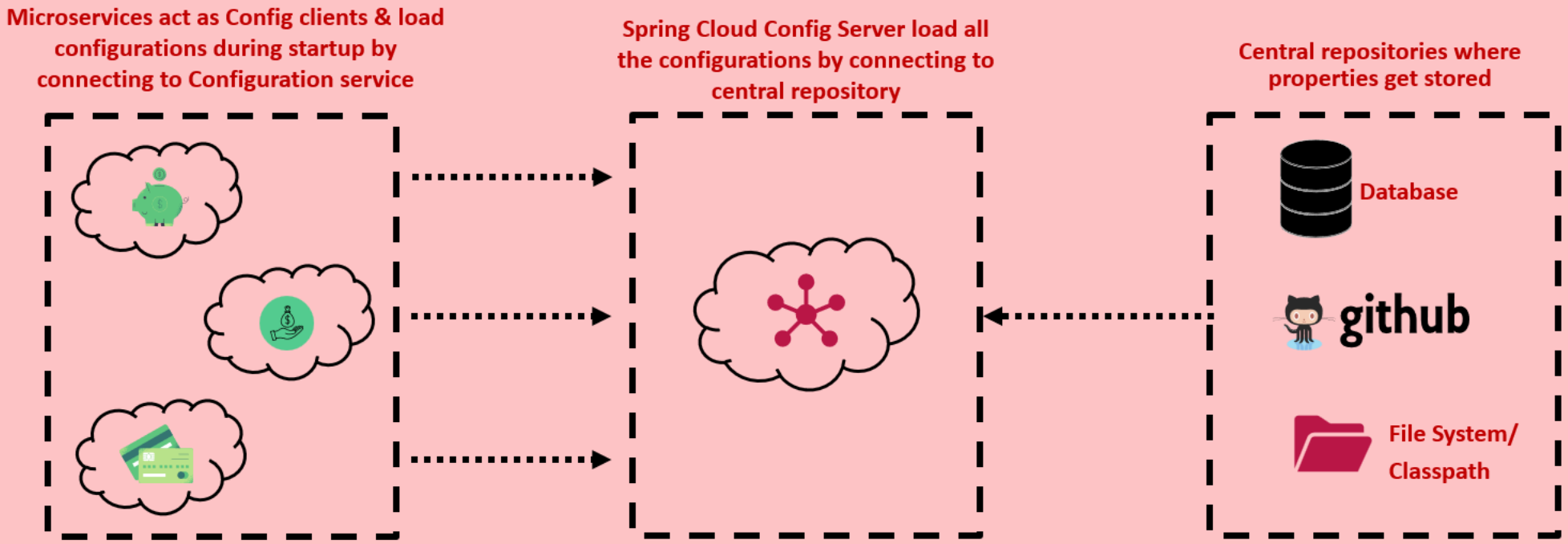
Spring Cloud Config

eazy bytes

A centralized configuration server with Spring Cloud Config can overcome all the drawbacks that we discussed in the previous slide. Spring Cloud Config provides server and client-side support for externalized configuration in a distributed system. With the Config Server you have a central place to manage external properties for applications across all environments.

Centralized configuration revolves around two core elements:

- A data store designed to handle configuration data, ensuring durability, version management, and potentially access control.
- A server that oversees the configuration data within the data store, facilitating its management and distribution to multiple applications.



WHAT IS SPRING CLOUD?

USING SPRING CLOUD FOR MICROSERVICES DEVELOPMENT

eazy bytes

Spring Cloud provides frameworks for developers to quickly build some of the common patterns of Microservices

ROUTING & TRACING

Makes sure that all calls to your microservices go through a single “front door” before the targeted service is invoked & the same will be traced.

LOAD BALANCING

Load balancing efficiently distributes network traffic to multiple backend servers or server pool

SERVICE REGISTRATION & DISCOVERY

New services will be registered & later consumers can invoke them through a logical name rather than physical location

SPRING CLOUD SECURITY

Provides features related to token-based security in Spring Boot applications/Microservices

SPRING CLOUD CONFIG

Ensures that no matter how many microservice instances you bring up; they’ll always have the same configuration.

DISTRIBUTED TRACING & MESSAGING

Incorporate components that helps in understanding the complex interactions between services and asynchronous communication, allowing for scalable and resilient systems.

Refresh configurations at runtime using /refresh path

eazy bytes

What occurs when new updates are committed to the Git repository supporting the Config Service? In a typical Spring Boot application, modifying a property would require a restart. However, Spring Cloud Config introduces the capability to dynamically refresh the configuration in client applications during runtime. When a change is pushed to the configuration repository, all integrated applications connected to the config server can be notified, prompting them to reload the relevant portions affected by the configuration modification.

Let's see an approach for refreshing the configuration, which involves sending a specific POST request to a running instance of the microservice. This request will initiate the reloading of the modified configuration data, enabling a hot reload of the application. Below are the steps to follow,

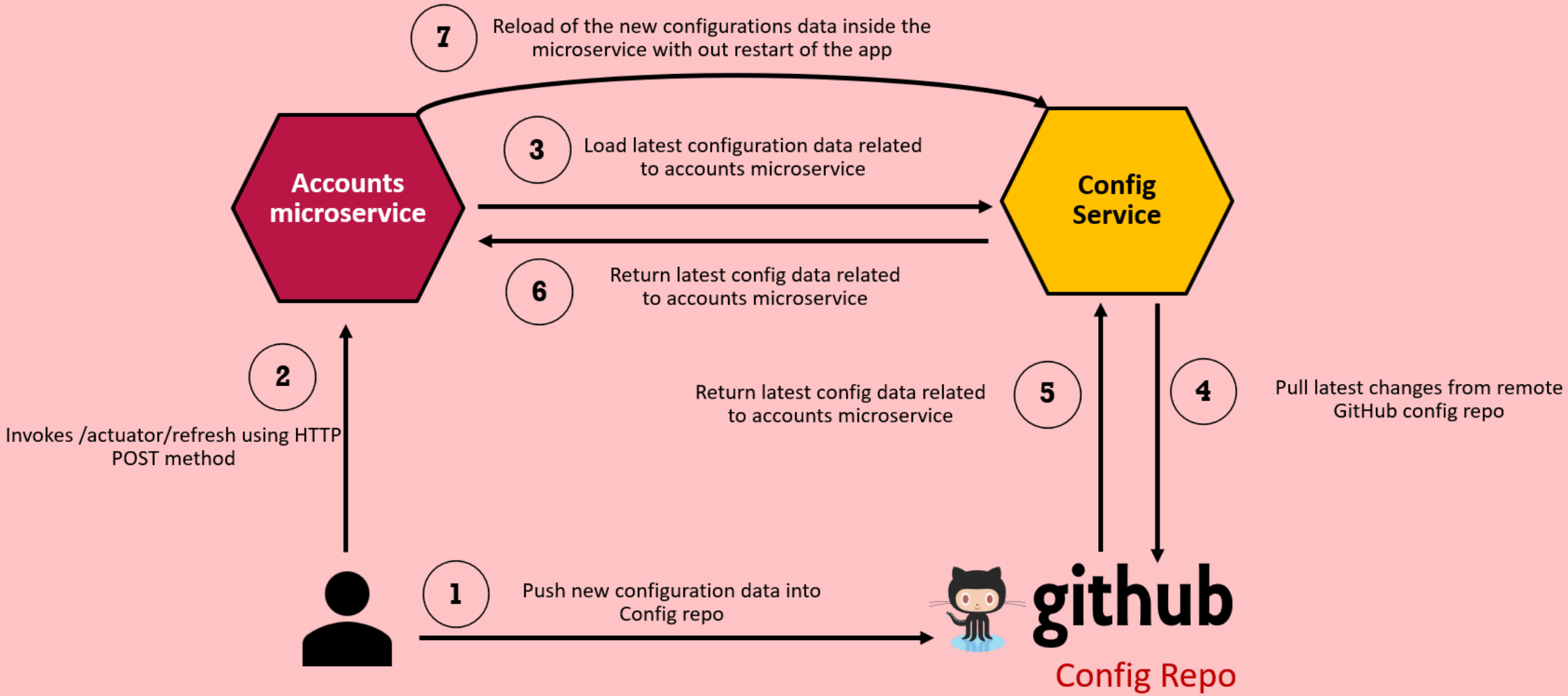
1. Add actuator dependency in the Config Client services: Add Spring Boot Actuator dependency inside pom.xml of the individual microservices like accounts, loans, cards to expose the /refresh endpoint

2. Enable /refresh API : The Spring Boot Actuator library provides a configuration endpoint called "/actuator/refresh" that can trigger a refresh event. By default, this endpoint is not exposed, so you need to explicitly enable it in the application.yml file using the below config,

```
management:
  endpoints:
    web:
      exposure:
        include: refresh
```

Refresh configurations at runtime using /refresh path

eazy bytes



You invoked the refresh mechanism on Accounts Service, and it worked fine, since it was just one application with 1 instance. How about in production where there may be multiple services ?If a project has many microservices, then team may prefer to have an automated and efficient method for refreshing configuration instead of manually triggering each application instance. Let's evaluate other options that we have

Refresh configurations at runtime using Spring Cloud Bus

eazy bytes

Spring Cloud Bus, available at https://spring.io/projects/spring-cloud-bus, facilitates seamless communication between all connected application instances by establishing a convenient event broadcasting channel. It offers an implementation for AMQP brokers, such as RabbitMQ, and Kafka, enabling efficient communication across the application ecosystem.

Below are the steps to follow,

1. **Add actuator dependency in the Config Server & Client services**: Add Spring Boot Actuator dependency inside pom.xml of the individual microservices like accounts, loans and cards to expose the /busrefresh endpoint

2. **Enable / busrefresh API** : The Spring Boot Actuator library provides a configuration endpoint called "/actuator/busrefresh" that can trigger a refresh event. By default, this endpoint is not exposed, so you need to explicitly enable it in the application.yml file using the below config,

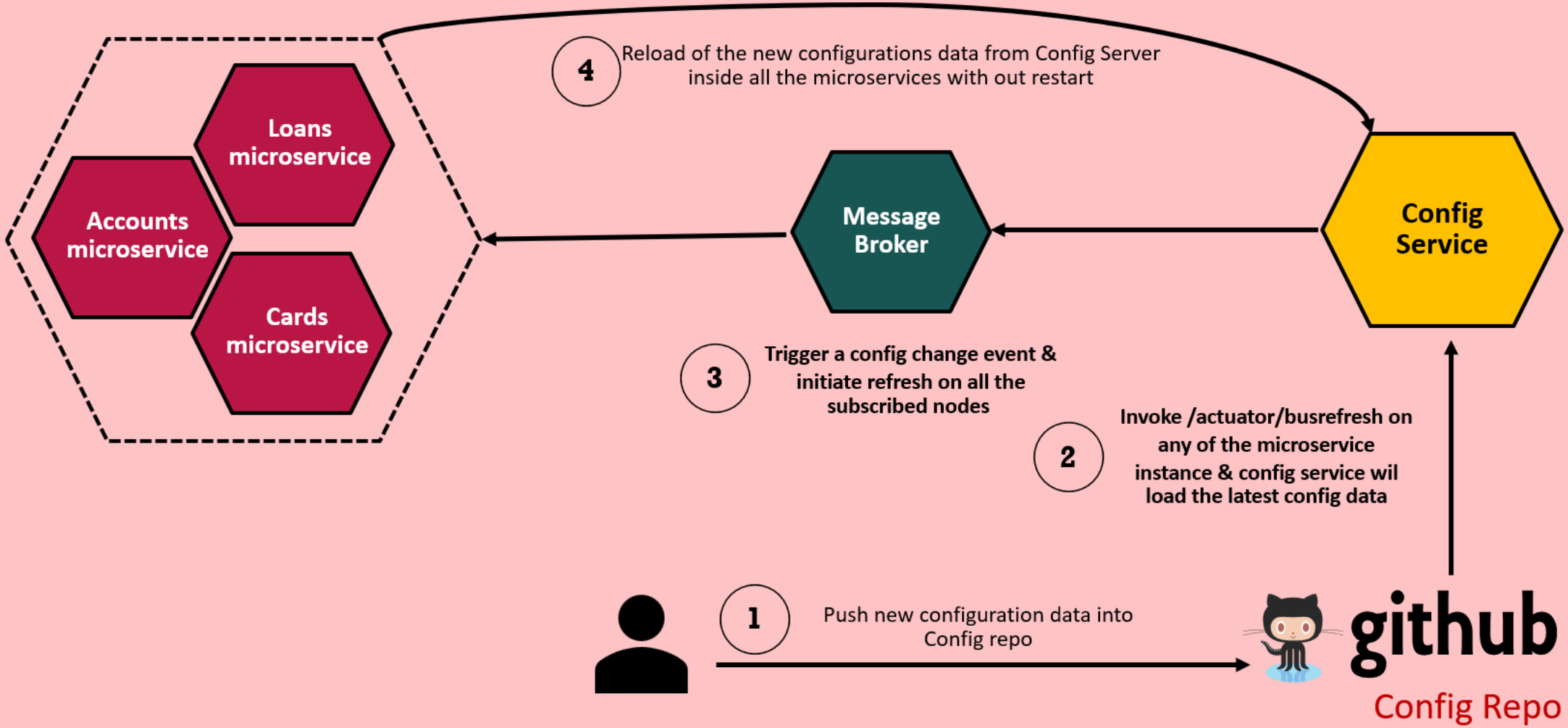
```
management:
  endpoints:
    web:
      exposure:
        include: busrefresh
```

3. **Add Spring Cloud Bus dependency in the Config Server & Client services**: Add Spring Cloud Bus dependency (spring-cloud-starter-bus-amqp) inside pom.xml of the individual microservices like accounts, loans, cards and Config server

4. **Set up a RabbitMQ**: Using Docker, setup RabbitMQ service. If the service is not started with default values, then configure the rabbitmq connection details in the application.yml file of all the individual microservices and Config server

Refresh configurations at runtime using Spring Cloud Bus

eazy bytes



Though this approach reduce manual work to a great extent, but still there is a single manual step involved which is invoking the /actuator/busrefresh on any of the microservice instance. Let's see how we can avoid and completely automate the process.

Refresh configurations at runtime using Spring Cloud Bus & Spring Cloud Config Monitor

eazy bytes

Spring Cloud Config offers the Monitor library, which enables the triggering of configuration change events in the Config Service. By exposing the /monitor endpoint, it facilitates the propagation of these events to all listening applications via the Bus. The Monitor library allows push notifications from popular code repository providers such as GitHub, GitLab, and Bitbucket. You can configure webhooks in these services to automatically send a POST request to the Config Service after each new push to the configuration repository. Below are the steps to follow,

1. **Add actuator dependency in the Config Server & Client services**: Add Spring Boot Actuator dependency inside pom.xml of the individual microservices like accounts, loans, cards and Config server to expose the /busrefresh endpoint

2. **Enable / busrefresh API** : The Spring Boot Actuator library provides a configuration endpoint called "/actuator/busrefresh" that can trigger a refresh event. By default, this endpoint is not exposed, so you need to explicitly enable it in the application.yml file using the below config,

```
management:
  endpoints:
    web:
      exposure:
        include: busrefresh
```

3. **Add Spring Cloud Bus dependency in the Config Server & Client services**: Add Spring Cloud Bus dependency (spring-cloud-starter-bus-amqp) inside pom.xml of the individual microservices like accounts, loans, cards and Config server

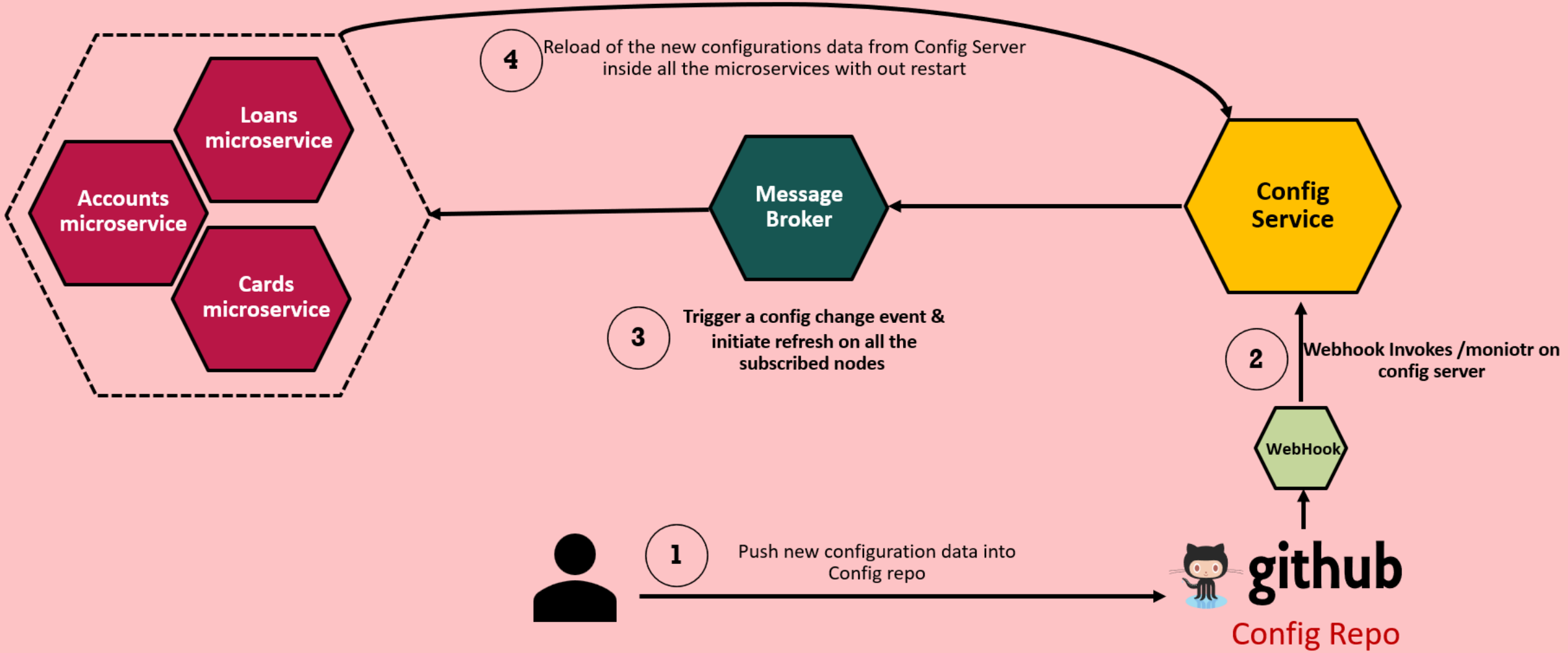
4. **Add Spring Cloud Config monitor dependency in the Config Server** : Add Spring Cloud Config monitor dependency (spring-cloud-config-monitor) inside pom.xml of Config server and this exposes /monitor endpoint

5. **Set up a RabbitMQ**: Using Docker, setup RabbitMQ service. If the service is not started with default values, then configure the rabbitmq connection details in the application.yml file of all the individual microservices and Config server

6. **Set up a WebHook in GitHub**: Set up a webhook to automatically send a POST request to Config Service /monitor path after each new push to the config repo.

Refresh configurations at runtime using Spring Cloud Bus & Spring Cloud Config Monitor

eazy bytes



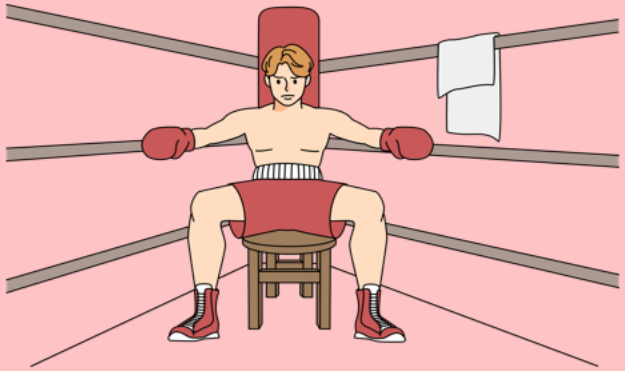
In this solution, there is no manual step involved and everything is automated.

Liveness and Readiness probes

eazy bytes

A **liveness** probe sends a signal that the container or application is either alive (passing) or dead (failing). If the container is alive, then no action is required because the current state is good. If the container is dead, then an attempt should be made to heal the application by restarting it.

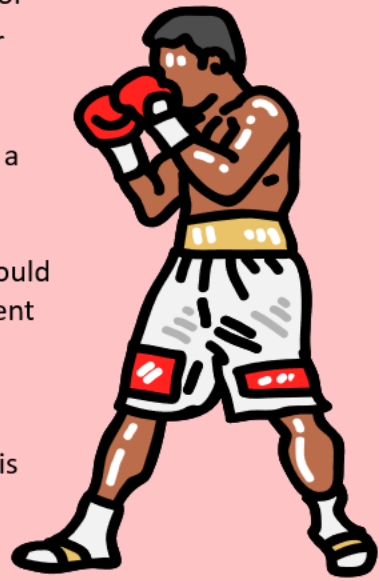
In simple words, liveness answers a true-or-false question: "Is this container alive?"



A **readiness** probe used to know whether the container or app being probed is ready to start receiving network traffic. If your container enters a state where it is still alive but cannot handle incoming network traffic (a common scenario during startup), you want the readiness probe to fail. So that, traffic will not be sent to a container which isn't ready for it.

If someone prematurely send network traffic to the container, it could cause the load balancer (or router) to return a 502 error to the client and terminate the request. The client would get a "connection refused" error message.

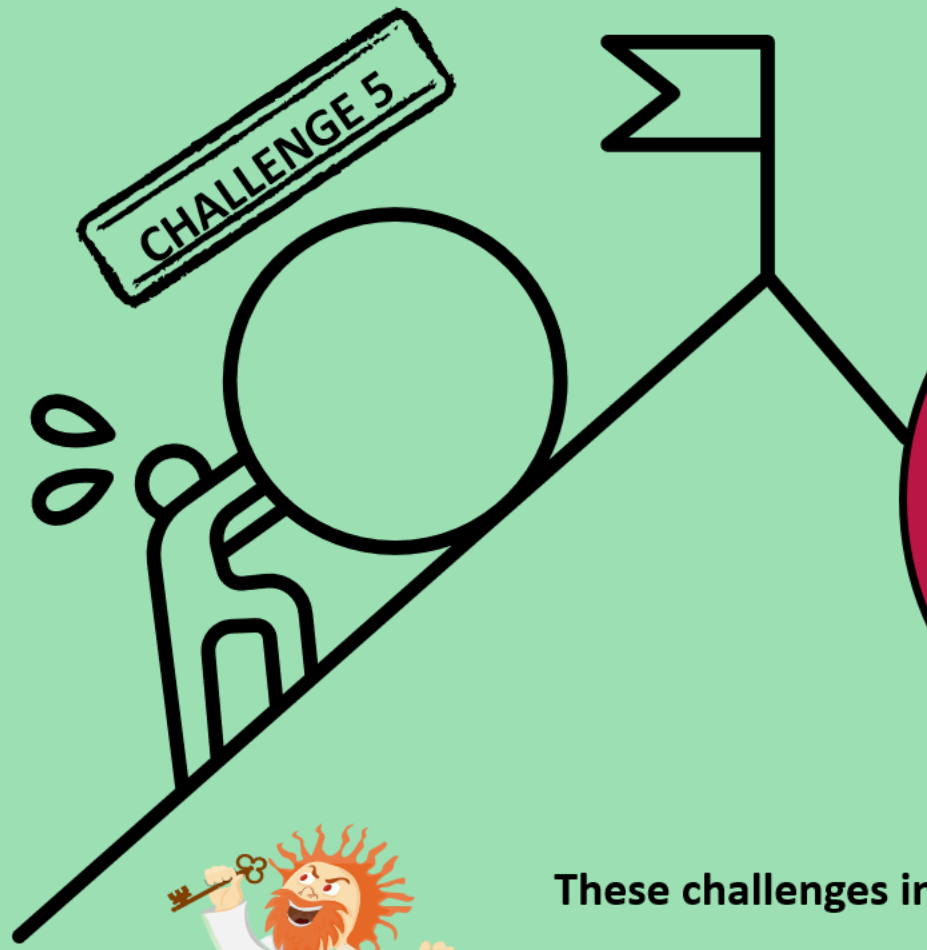
In simple words, readiness answers a true-or-false question: "Is this container ready to receive network traffic ?"



Inside Spring Boot apps, actuator gathers the “Liveness” and “Readiness” information from the ApplicationAvailability interface and uses that information in dedicated health indicators: LivenessStateHealthIndicator and ReadinessStateHealthIndicator. These indicators are shown on the global health endpoint ("/actuator/health"). They are also exposed as separate HTTP Probes by using health groups: "/actuator/health/liveness" and "/actuator/health/readiness"

SERVICE DISCOVERY & REGISTRATION IN MICROSERVICES

eazy bytes



HOW DO SERVICES LOCATE EACH OTHER INSIDE A NETWORK?

Each instance of a microservice exposes a remote API with it's own host and port. how do other microservices & clients know about these dynamic endpoint URLs to invoke them. So where is my service?

HOW DO NEW SERVICE INSTANCES ENTER INTO THE NETWORK?

If an microservice instance fails, new instances will be brought online to ensure constant availability. This means that the IP addresses of the instances can be constantly changing. So how does these new instances can start serving to the clients?

LOAD BALANCE, INFO SHARING B/W MICROSERVICE INSTANCES

How do we make sure to properly load balance b/w the multiple microservice instances especially a microservice is invoking another microservice? How do a specific service information shared across the network?

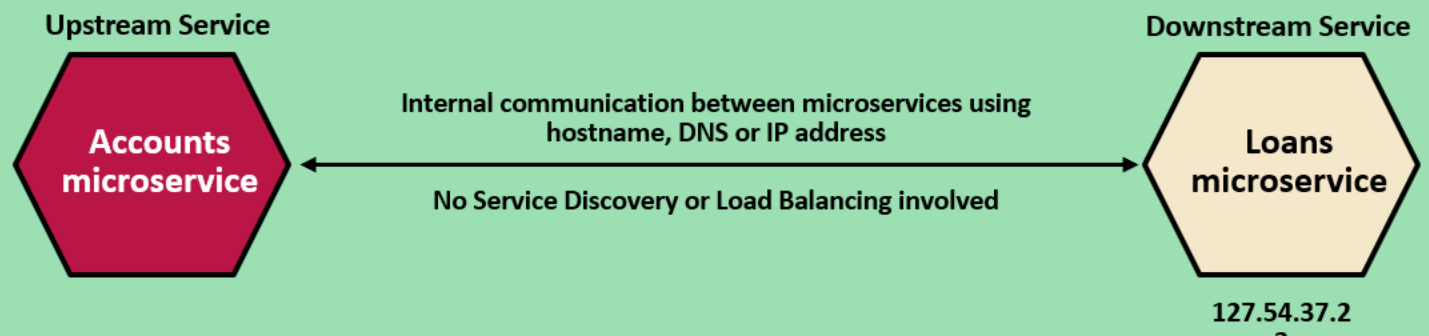
These challenges in microservices can be solved using below concepts or solutions,

1) **Service discovery**
2) **Service registration**
3) **Load balancing**

How service communication happens in Traditional apps ?

eazy bytes

Inside web network, When a service/app want to communicate with another service/app, it must be given the necessary information to locate it, such as an IP address or a DNS name. Let's examine the scenario of two services, Accounts and Loans. If there was only a single instance of Loans microservice, below figure illustrates how the communication between the two applications would occur.



Loans microservice will be a backing service with respect to Accounts microservice

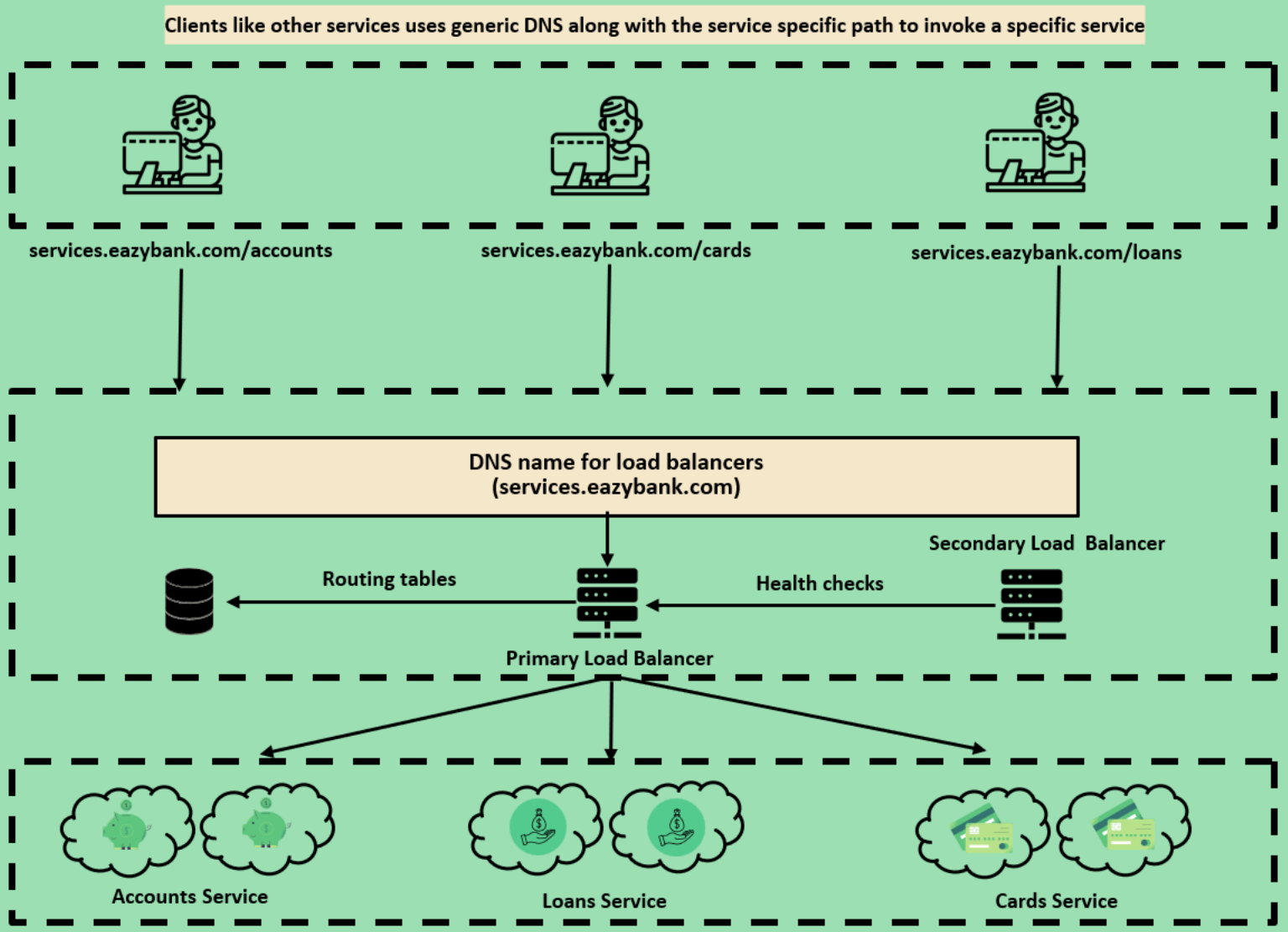
When there is only one instance of the Loans microservice running, managing the DNS name and its corresponding IP address mapping is straightforward. However, in a cloud environment, it is common to deploy multiple instances of a service, with each instance having its own unique IP address.

To address this challenge, one approach is to update DNS records with multiple IP addresses and rely on round-robin name resolution. This method directs requests to one of the IP addresses assigned to the service replicas in a rotating manner. However, this approach may not be suitable for microservices, as containers or services change frequently. This rapid change makes it difficult to maintain accurate DNS records and ensure efficient communication between microservices.

Unlike physical machines or long-running virtual machines, cloud-based service instances have shorter lifespans. These instances are designed to be disposable and can be terminated or replaced for various reasons, such as unresponsiveness. Furthermore, auto-scaling capabilities can be enabled to automatically adjust the number of application instances based on the workload.

How Traditional LoadBalancers works ?

eazy bytes



Traditional Service location resolution architecture using DNS & a load balancer

Limitations with Traditional LoadBalancers ?

eazy bytes

With traditional approach each instance of a service used to be deployed in one or more application servers. The number of these application servers was often static and even in the case of restoration it would be restored to the same state with the same IP and other configurations.

While this type of model works well with monolithic and SOA based applications with a relatively small number of services running on a group of static servers, it doesn't work well for cloud-based microservice applications for the following reasons,

- Limited horizontal scalability & licenses costs
- Single point of failure & Centralized chokepoints
- Manually managed to update any IPs, configurations
- Complex in nature & not containers friendly

The biggest challenge with traditional load balancers is that some one has to manually maintain the routing tables which is an impossible task inside the microservices network. Because containers/services are ephemeral in nature

How to solve the problem for cloud native applications ?

eazy bytes

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 application 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.

Client-side service discovery and server-side service discovery are distinct approaches that address the service discovery problem in different contexts

How to solve the problem for cloud native applications ?

eazy bytes

In a modern microservice architecture, knowing the right network location of an application is a much more complex problem for the clients as service instances might have dynamically assigned IP addresses. Moreover the number instances may vary due to autoscaling and failures.

Microservices service discovery & registration is a way for applications and microservices to locate each other on a network. This includes,

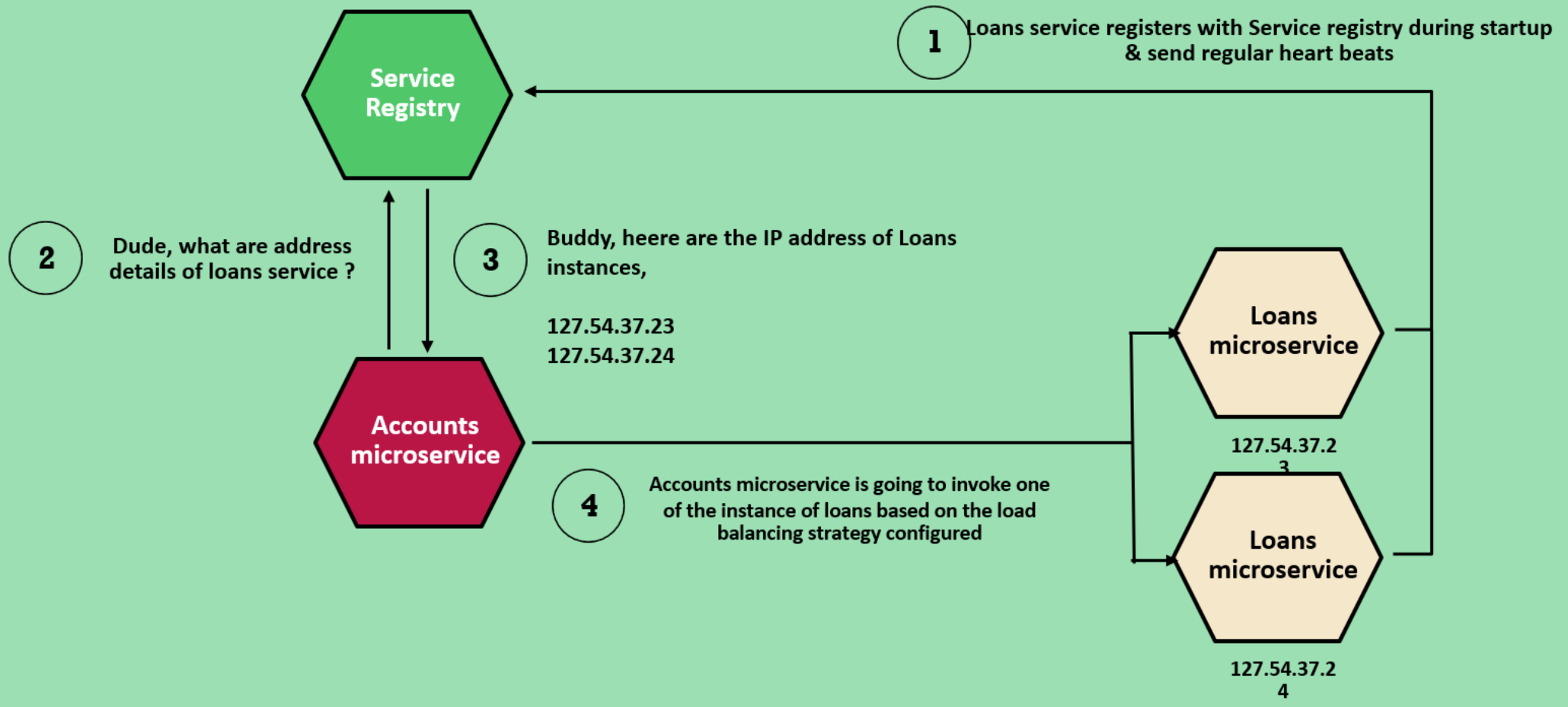
- A central server (or servers) that maintain a global view of addresses
- Microservices/clients that connect to the central server to register their address when they start & ready
- Microservices/clients need to send their heartbeats at regular intervals to central server about their health
- Microservices/clients that connect to the central server to deregister their address when they are about to shutdown

Service discovery & registrations deals with the problems about how microservices talk to each other, i.e. perform API calls.

Client-side service discovery and load balancing

eazy bytes

In client-side service discovery, applications are responsible for registering themselves with a service registry during startup and unregistering when shutting down. When an application needs to communicate with a backing service, it queries the service registry for the associated IP address. If multiple instances of the service are available, the registry returns a list of IP addresses. The client application then selects one based on its own defined load-balancing strategy. Below figure illustrates the workflow of this process



Client-side service discovery and load balancing

eazy bytes

Client-side service discovery is an architectural pattern where client applications are responsible for locating and connecting to services they depend on. In this approach, the client application communicates directly with a service registry to discover available service instances and obtain the necessary information to establish connections.

Here are the key aspects of client-side service discovery:

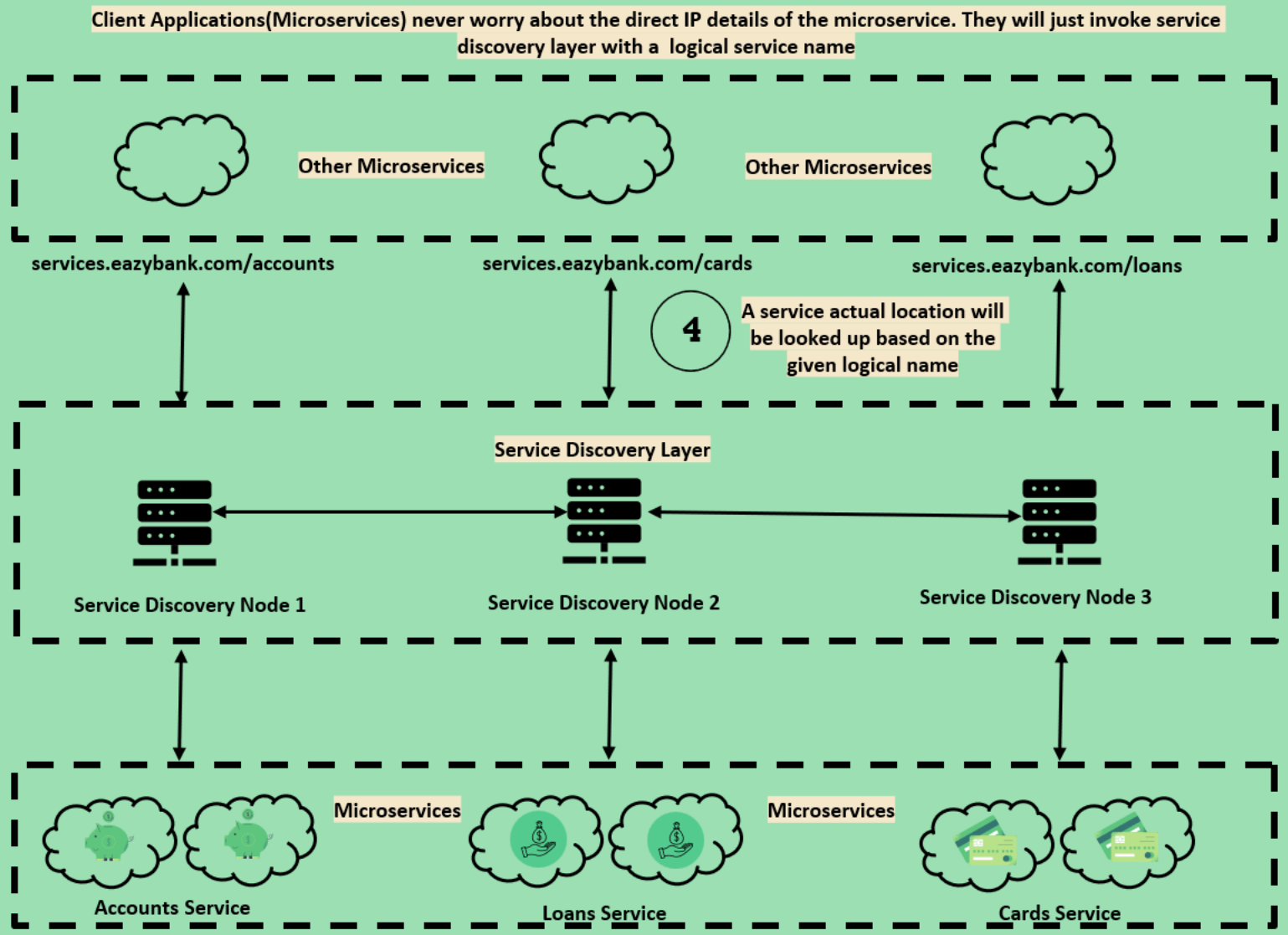
- **Service Registration:** Client applications register themselves with the service registry upon startup. They provide essential information about their location, such as IP address, port, and metadata, which helps identify and categorize the service.
- **Service Discovery:** When a client application needs to communicate with a specific service, it queries the service registry for available instances of that service. The registry responds with the necessary information, such as IP addresses and connection details.
- **Load Balancing:** Client-side service discovery often involves load balancing to distribute the workload across multiple service instances. The client application can implement a load-balancing strategy to select a specific instance based on factors like round-robin, weighted distribution, or latency.

The major advantage of client-side service discovery is load balancing can be implemented using various algorithms, such as round-robin, weighted round-robin, least connections, or even custom algorithms. A drawback is that client service discovery assigns more responsibility to developers. Also, it results in one more service to deploy and maintain (the service registry). Server-side discovery solutions solve these issues. We are going to discuss the same when we are talking about Kubernetes

The Spring Cloud project provides several alternatives for incorporating client-side service discovery in our Spring Boot based microservices. More details to follow...

How Client-side service discovery works ?

eazy bytes

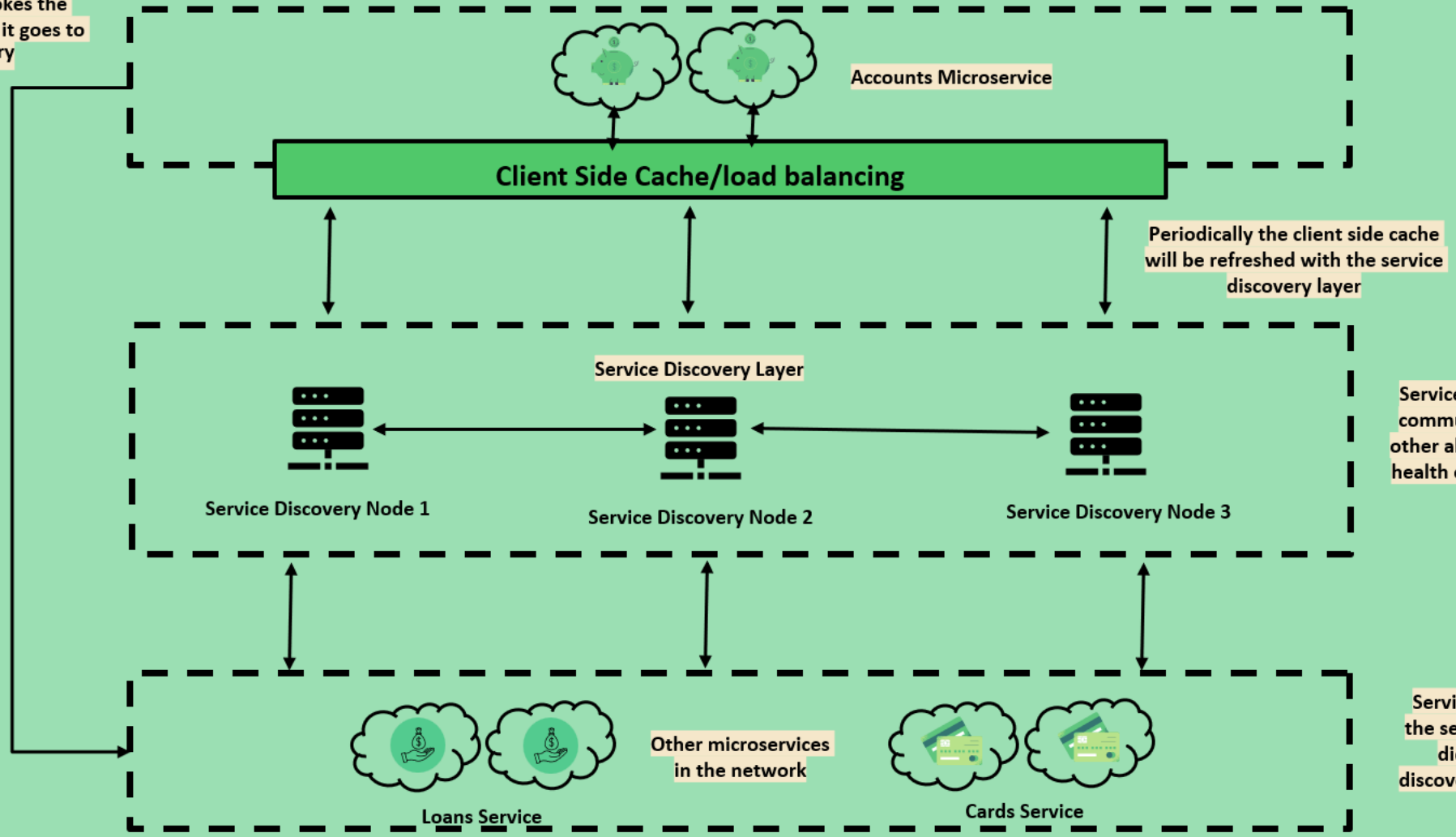


How loadbalancing works in Client-side service discovery ?

eazy bytes

When a microservice want to connect with other microservice, it will check the local cache for the service instances IPs. Load balancing also happens at the service level itself w/o depending on the Service Discovery

If the client finds a service IP in the cache, it will directly invokes the backing service. Otherwise it goes to the service discovery

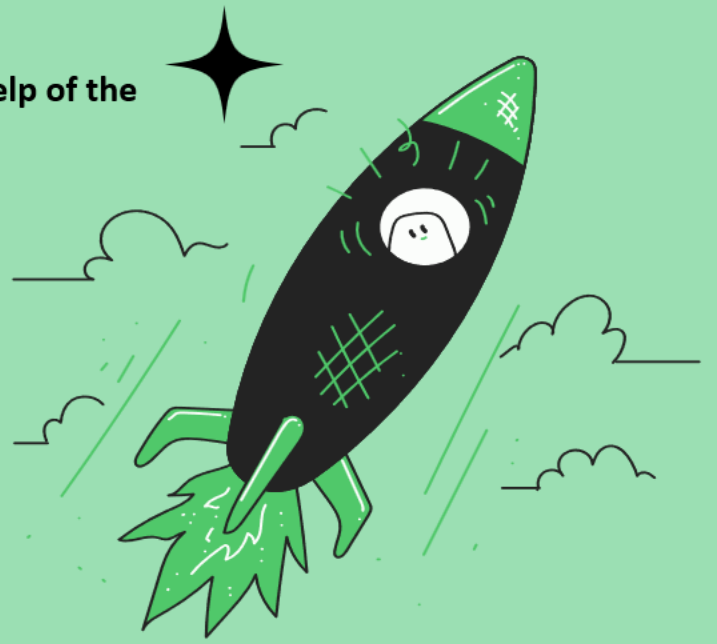


Spring Cloud support for Client-side service discovery

eazy bytes

Spring Cloud project makes Service Discovery & Registration setup trivial to undertake with the help of the below components,

- **Spring Cloud Netflix's Eureka** service which will act as a service discovery agent
- **Spring Cloud Load Balancer** library for client-side load balancing
- **Netflix Feign client** to look up for a service b/w microservices



Though in this course we use Eureka since it is mostly used but they are other service registries such as etcd,Consul, and Apache Zookeeper which are also good.

Though Netflix Ribbon client-side is also good and stable product, we are going to use Spring Cloud Load Balancer for client-side load balancing. This is because Ribbon has entered a maintenance mode and unfortunately, it will not be developed anymore

Advantages of Service Discovery approach includes,

- No limitations on availability
- Peer to peer communication b/w Services Discovery agents
- Dynamically managed IPs, configurations & Load balanced
- Fault-tolerant & Resilient in nature

Steps to build Eureka Server

eazy bytes

Below are the steps to build a Eureka Server application using Spring Cloud Netflix's Eureka,

1. **Set up a new Spring Boot project:** Start by creating a new Spring Boot project using your preferred IDE or by using Spring Initializr (https://start.spring.io/). Include the spring-cloud-starter-netflix-eureka-server maven dependency.

2. **Configure the properties:** In the application properties or YAML file, add the following configurations,

```
server:
  port: 8070

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

3. **Add the Eureka Server annotation:** In the main class of your project, annotate it with @EnableEurekaServer. This annotation configures the application to act as a Eureka Server.

4. **Build and run the Eureka Server:** Build your project and run it as a Spring Boot application. Open a web browser and navigate to http://localhost:8070. You should see the Eureka Server dashboard, which displays information about registered service instances.

Steps to register a microservice as a Eureka Client

eazy bytes

Below are the steps to make a microservice application to register and act as a Eureka client,

1. **Set up a new Spring Boot project:** Start by creating a new Spring Boot project using your preferred IDE or by using Spring Initializr (https://start.spring.io/). Include the spring-cloud-starter-netflix-eureka-client maven dependency.

2. **Configure the properties:** In the application properties or YAML file, add the following configurations,

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

3. **Build and run the application:** Build your project and run it as a Spring Boot application. Open a web browser and navigate to http://localhost:8070. You should see the microservice registered itself as an application and the same can be confirmed inside the Eureka Server dashboard.

EUREKA SELF-PRESERVATION TO AVOID TRAPS IN NETWORK

eazy bytes

In a distributed system using Eureka, each service instance periodically sends a heartbeat signal to the Eureka server to indicate that it is still alive and functioning. If the Eureka server does not receive a heartbeat from a service instance within a certain timeframe, it assumes that the instance has become unresponsive or has crashed. In normal scenarios, this behavior helps the Eureka server maintain an up-to-date view of the registered service instances.

However, in certain situations, network glitches or temporary system delays may cause the Eureka server to miss a few heartbeats, leading to false expiration of service instances. This can result in unnecessary evictions of healthy service instances from the registry, causing instability and disruption in the system.

To mitigate this issue, Eureka enters into Self-Preservation mode. When Self-Preservation mode is active, the existing registry entries will not be removed even if it stops receiving heartbeats from some of the service instances. This prevents the Eureka server from evicting all the instances due to temporary network glitches or delays.

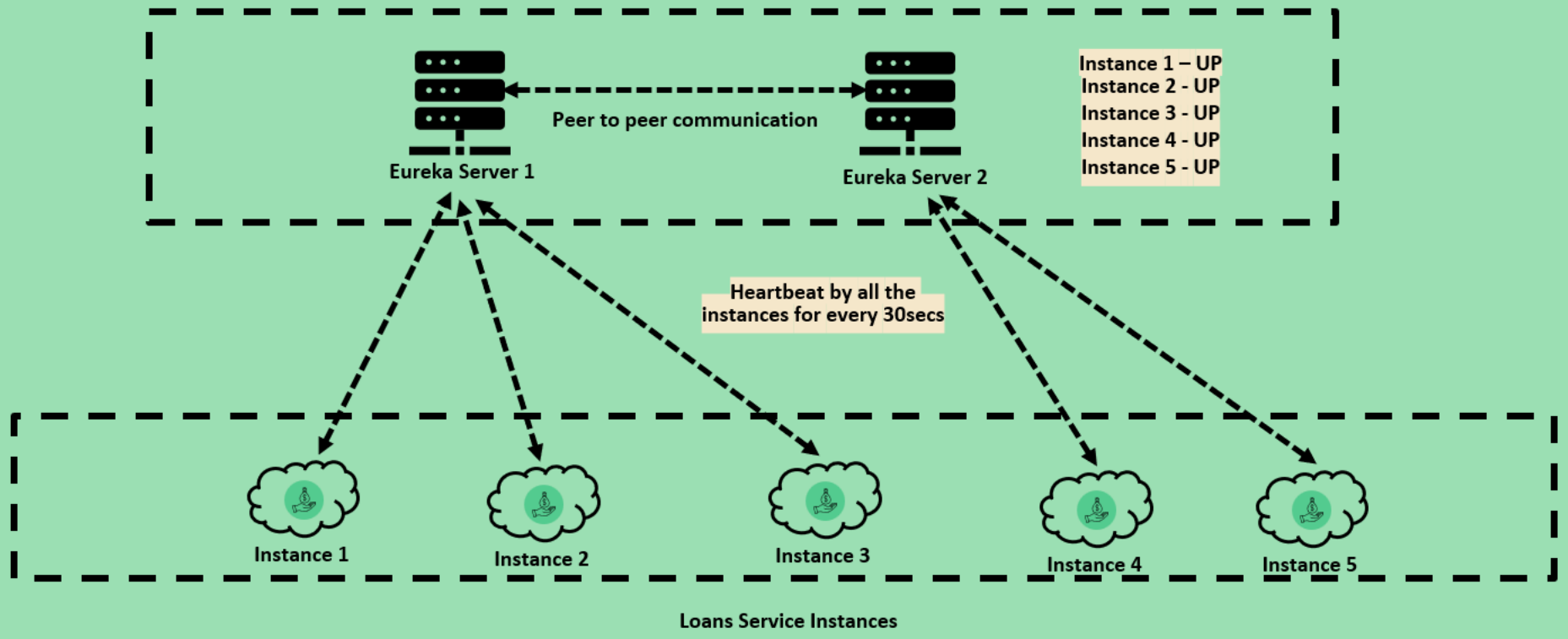
In Self-Preservation mode, the Eureka server continues to serve the registered instances to client applications, even if it suspects that some instances are no longer available. This helps maintain the stability and availability of the service registry, ensuring that clients can still discover and interact with the available instances.

Self-preservation mode never expires, until and unless the down microservices are brought back or the network glitch is resolved. This is because eureka will not expire the instances till it is above the threshold limit.

Eureka Server will not panic when it is not receiving heartbeats from majority of the instances, instead it will be calm and enters into Self-preservation mode. This feature is a savior where the networks glitches are common and help us to handle false-positive alarms.

EUREKA SELF-PRESERVATION TO AVOID TRAPS IN NETWORK

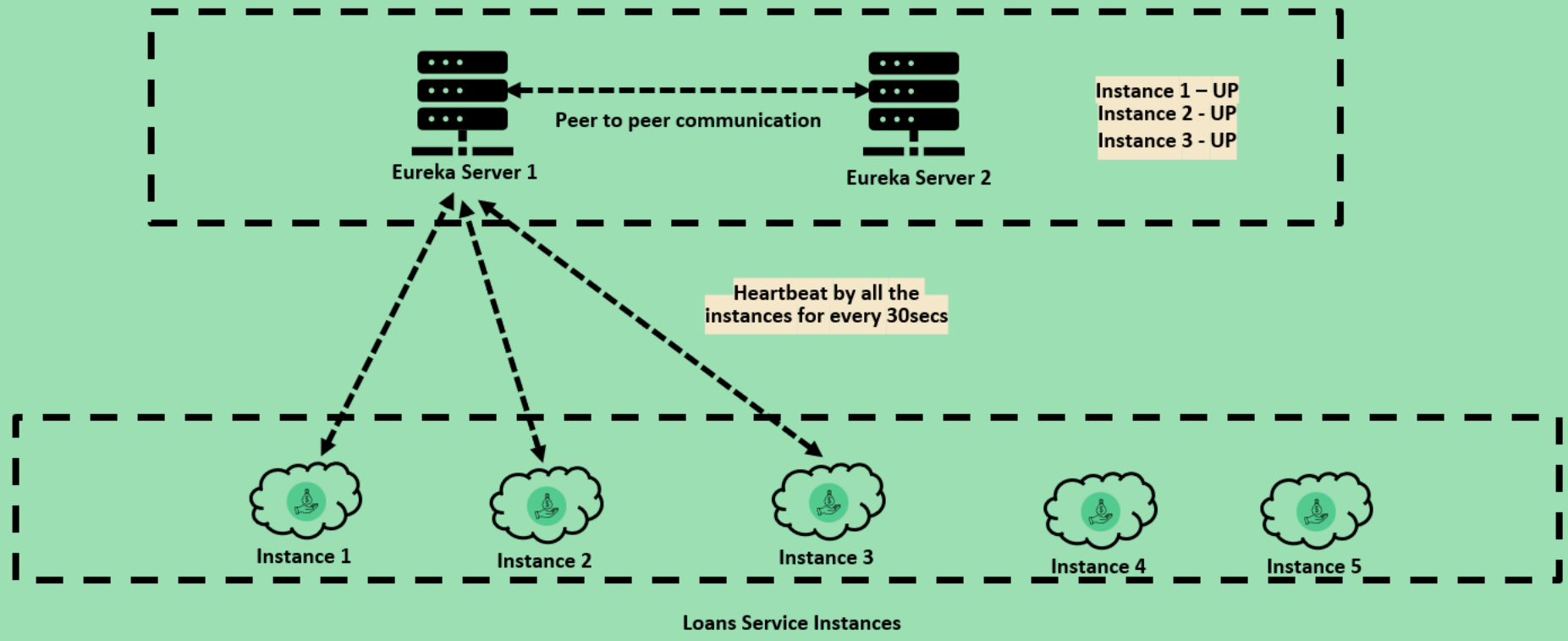
eazy bytes



Healthy Microservices System with all 5 instances up before encountering network problems

EUREKA SELF-PRESERVATION TO AVOID TRAPS IN NETWORK

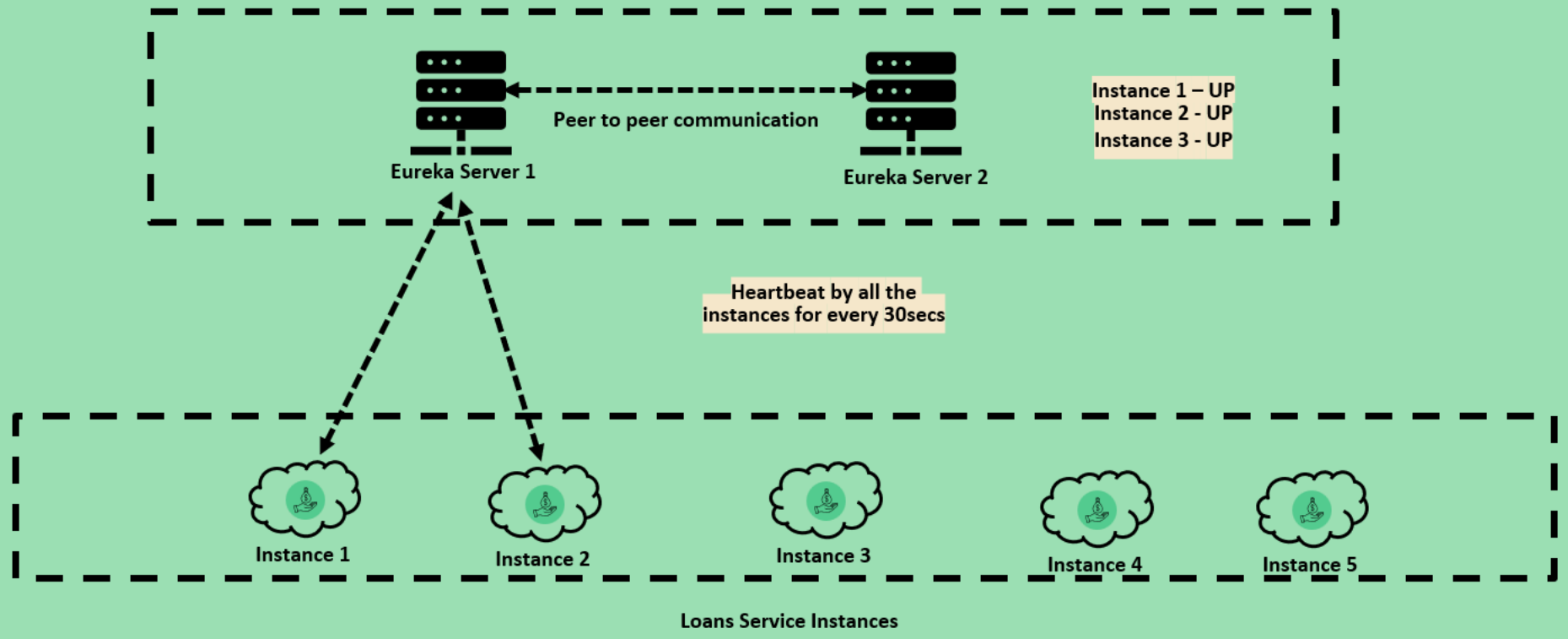
eazy bytes



2 of the instances not sending heartbeat. Eureka enters self-preservation mode since it met threshold percentage

EUREKA SELF-PRESERVATION TO AVOID TRAPS IN NETWORK

eazy bytes



During Self-preservation, eureka will stop expiring the instances though it is not receiving heartbeat from instance 3

EUREKA SELF-PRESERVATION TO AVOID TRAPS IN NETWORK

eazy bytes

- Configurations which will directly or indirectly impact self-preservation behavior of eureka

✓ `eureka.instance.lease-renewal-interval-in-seconds = 30`
Indicates the frequency the client sends heartbeats to server to indicate that it is still alive

✓ `eureka.instance.lease-expiration-duration-in-seconds = 90`
Indicates the duration the server waits since it received the last heartbeat before it can evict an instance

✓ `eureka.server.eviction-interval-timer-in-ms = 60 * 1000`
A scheduler(EvictionTask) is run at this frequency which will evict instances from the registry if the lease of the instances are expired as configured by lease-expiration-duration-in-seconds. It will also check whether the system has reached self-preservation mode (by comparing actual and expected heartbeats) before evicting.

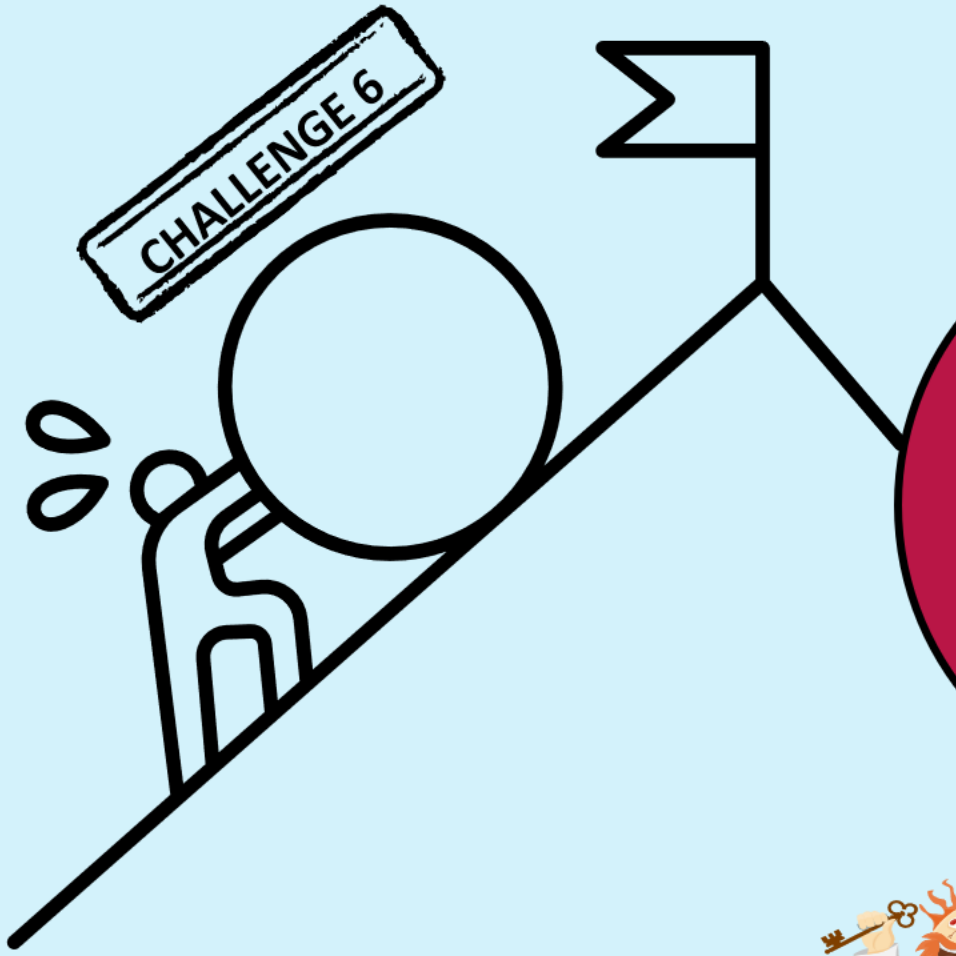
✓ `eureka.server.renewal-percent-threshold = 0.85`
This value is used to calculate the expected % of heartbeats per minute eureka is expecting.

✓ `eureka.server.renewal-threshold-update-interval-ms = 15 * 60 * 1000`
A scheduler is run at this frequency which calculates the expected heartbeats per minute

✓ `eureka.server.enable-self-preservation = true`
By default self-preservation mode is enabled but if you need to disable it you can change it to ‘false’

ROUTING, CROSS CUTTING CONCERNS IN MICROSERVICES

eazy bytes



HOW DO WE MAINTAIN A SINGLE ENTRYPOINT INTO MICROSERVICES NETWORK ?

How do we build a single gatekeeper for all the inbound traffic to our microservices. This way the client doesn't need to keep track of the different services involved in a transaction, simplifying the client's logic

HOW DO WE HANDLE CROSS CUTTING CONCERNS?

In a distributed microservices architecture, how do we make sure to have a consistently enforced cross cutting concerns like logging, auditing, tracing and security across multiple microservices

HOW DO WE ROUTE BASED ON CUSTOM REQUIREMENTS

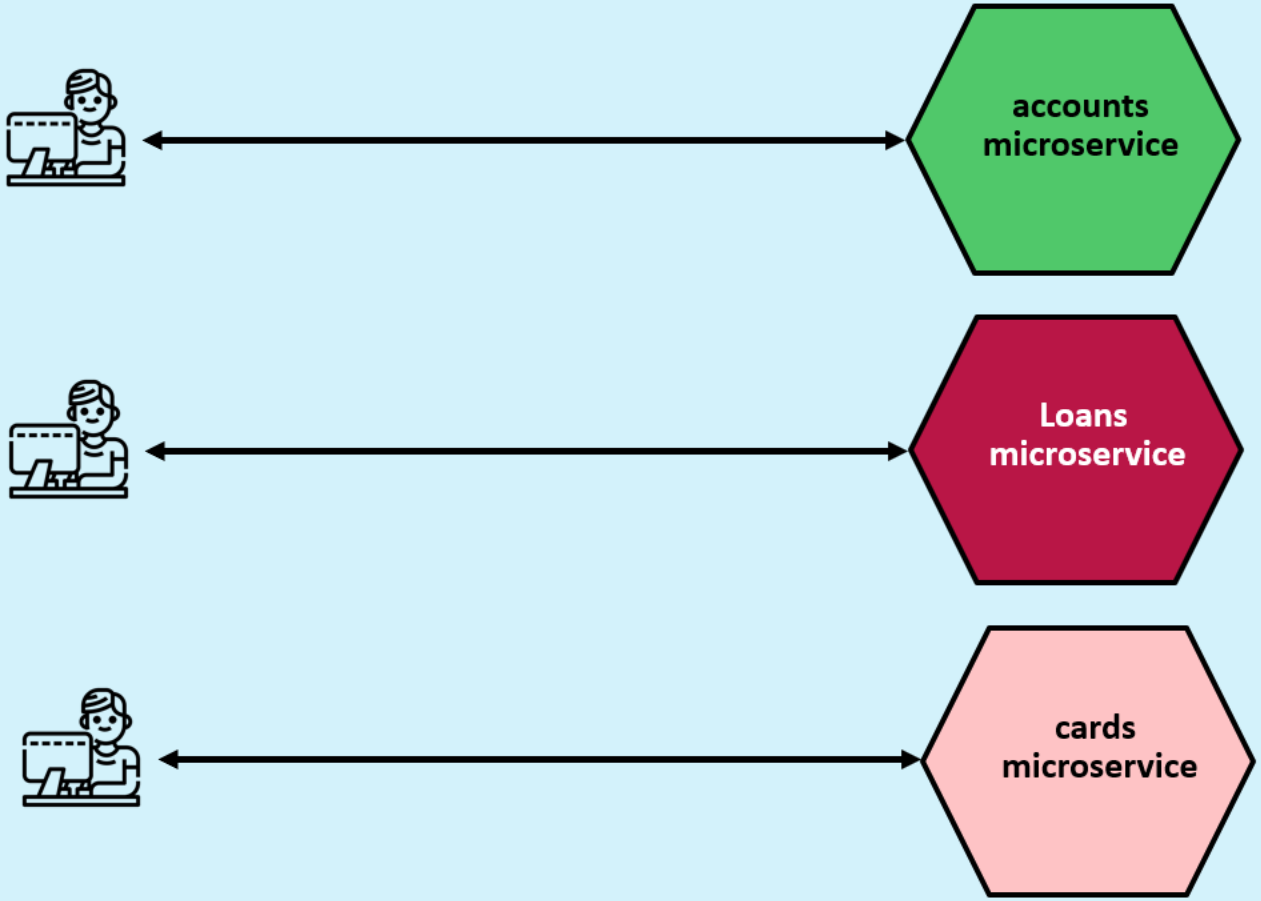
How to provide dynamic routing capabilities which allows to define routing rules based on various criteria such as HTTP headers, request parameters etc. inside microservices network ?

These challenges in microservices can be solved using a

Edge server

ROUTING, CROSS CUTTING CONCERNS IN MICROSERVICES

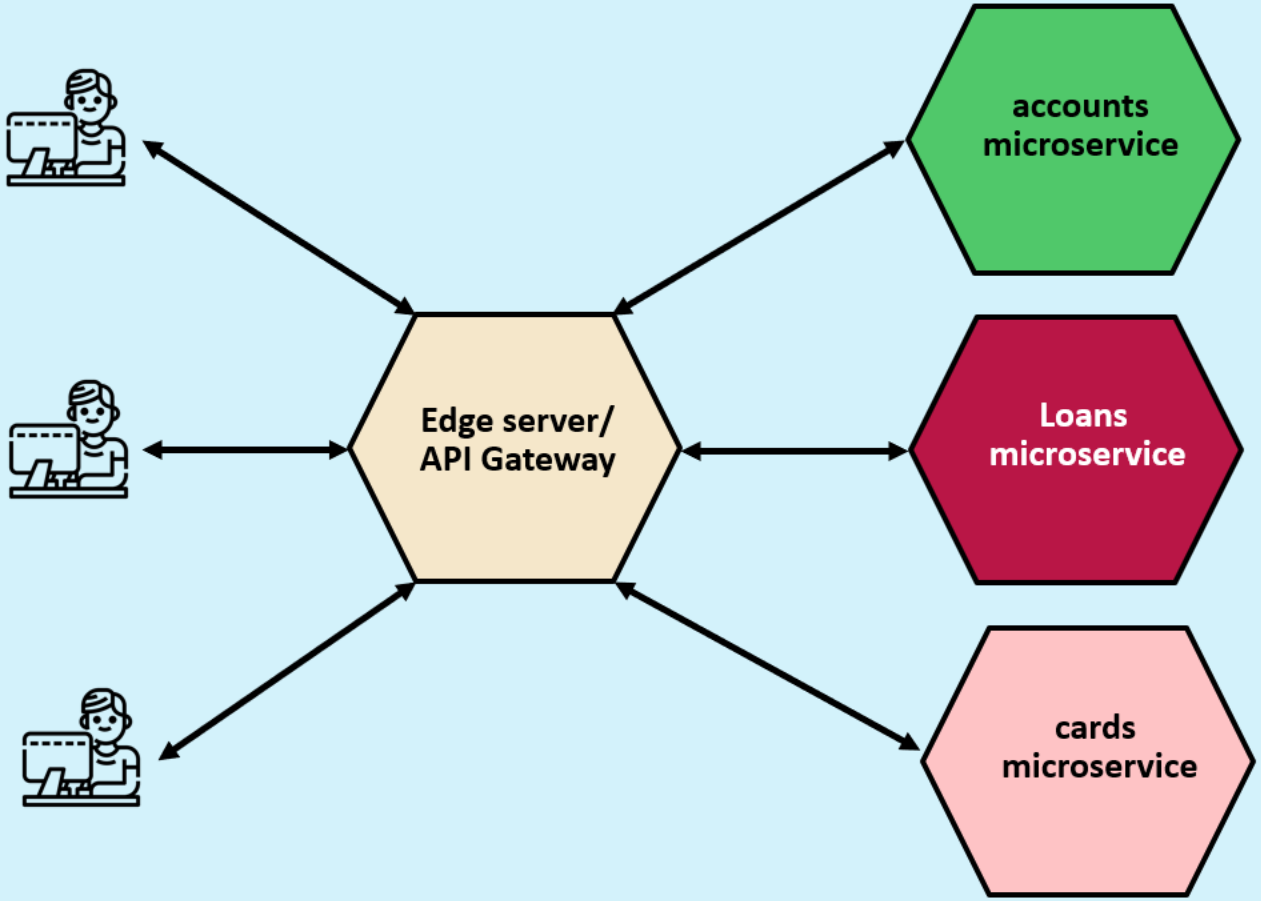
eazy bytes



In a scenario where multiple clients directly connect with various services, several challenges arise. For instance, clients must be aware of the URLs of all the services, and enforcing common requirements such as security, auditing, logging, and routing becomes a repetitive task across all services. To address these challenges, it becomes necessary to establish a single gateway as the entry point to the microservices network.

ROUTING, CROSS CUTTING CONCERNS IN MICROSERVICES

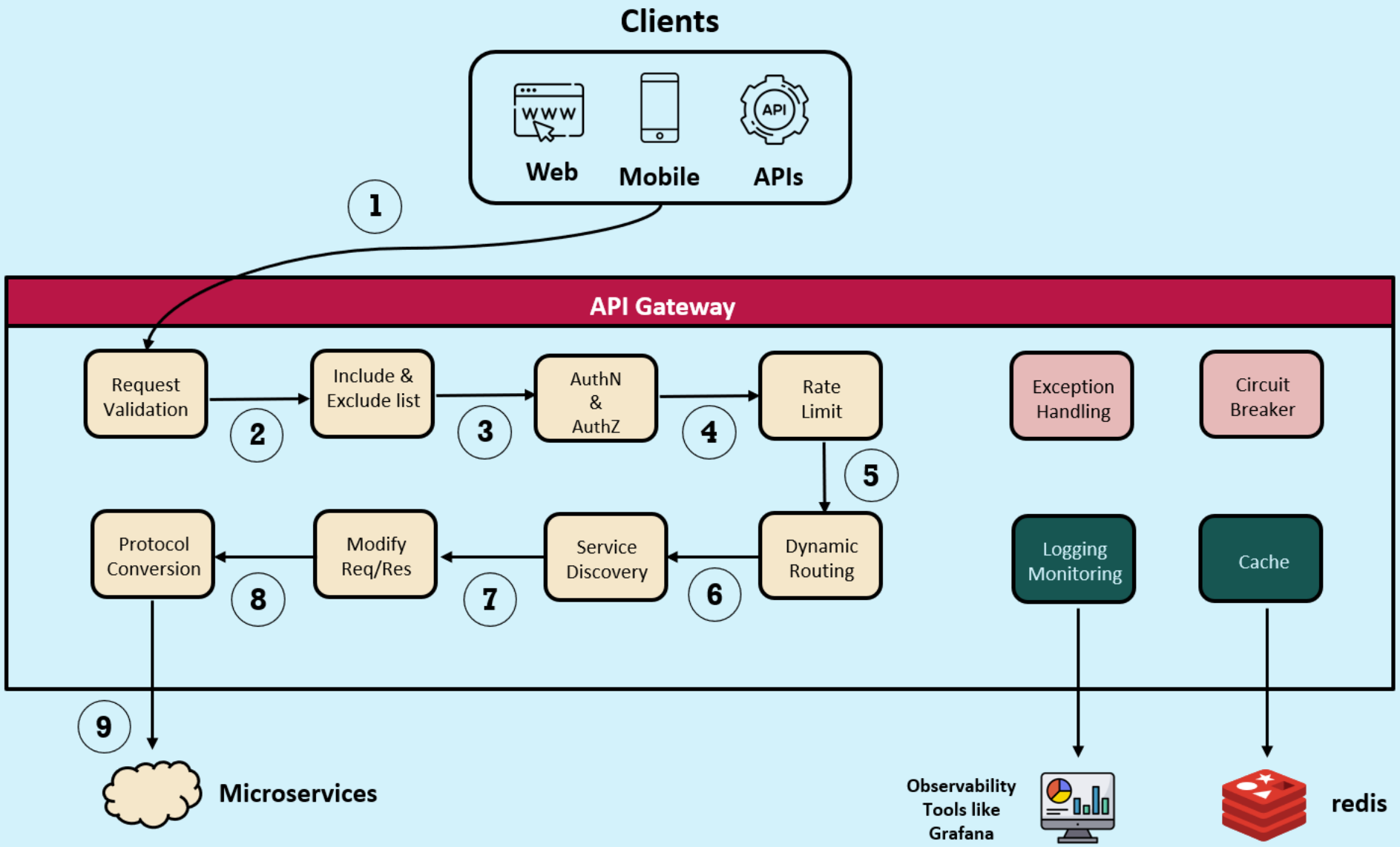
eazy bytes



Edge servers are applications positioned at edge of a system, responsible for implementing functionalities such as API gateways and handling cross-cutting concerns. By utilizing edge servers, it becomes possible to prevent cascading failures when invoking downstream services, allowing for the specification of retries and timeouts for all internal service calls. Additionally, these servers enable control over ingress traffic, empowering the enforcement of quota policies. Furthermore, authentication and authorization mechanisms can be implemented at the edge, enabling the passing of tokens to downstream services for secure communication and access control.

Few important tasks that API Gateway does

eazy bytes



Spring Cloud Gateway

eazy bytes

Spring Cloud Gateway streamlines the creation of edge services by emphasizing ease and efficiency. Moreover, due to its utilization of a reactive framework, it can seamlessly expand to handle the significant workload that typically arises at the system's edge while maintaining optimal scalability.

Here are the key aspects of Spring Cloud Gateway,

- The service gateway sits as the gatekeeper for all inbound traffic to microservice calls within our application. With a service gateway in place, our service clients never directly call the URL of an individual service, but instead place all calls to the service gateway.
- Spring Cloud Gateway is a library for building an API gateway, so it looks like any another Spring Boot application. If you're a Spring developer, you'll find it's very easy to get started with Spring Cloud Gateway with just a few lines of code.
- Spring Cloud Gateway is intended to sit between a requester and a resource that's being requested, where it intercepts, analyzes, and modifies every request. That means you can route requests based on their context. Did a request include a header indicating an API version? We can route that request to the appropriately versioned backend. Does the request require sticky sessions? The gateway can keep track of each user's session.

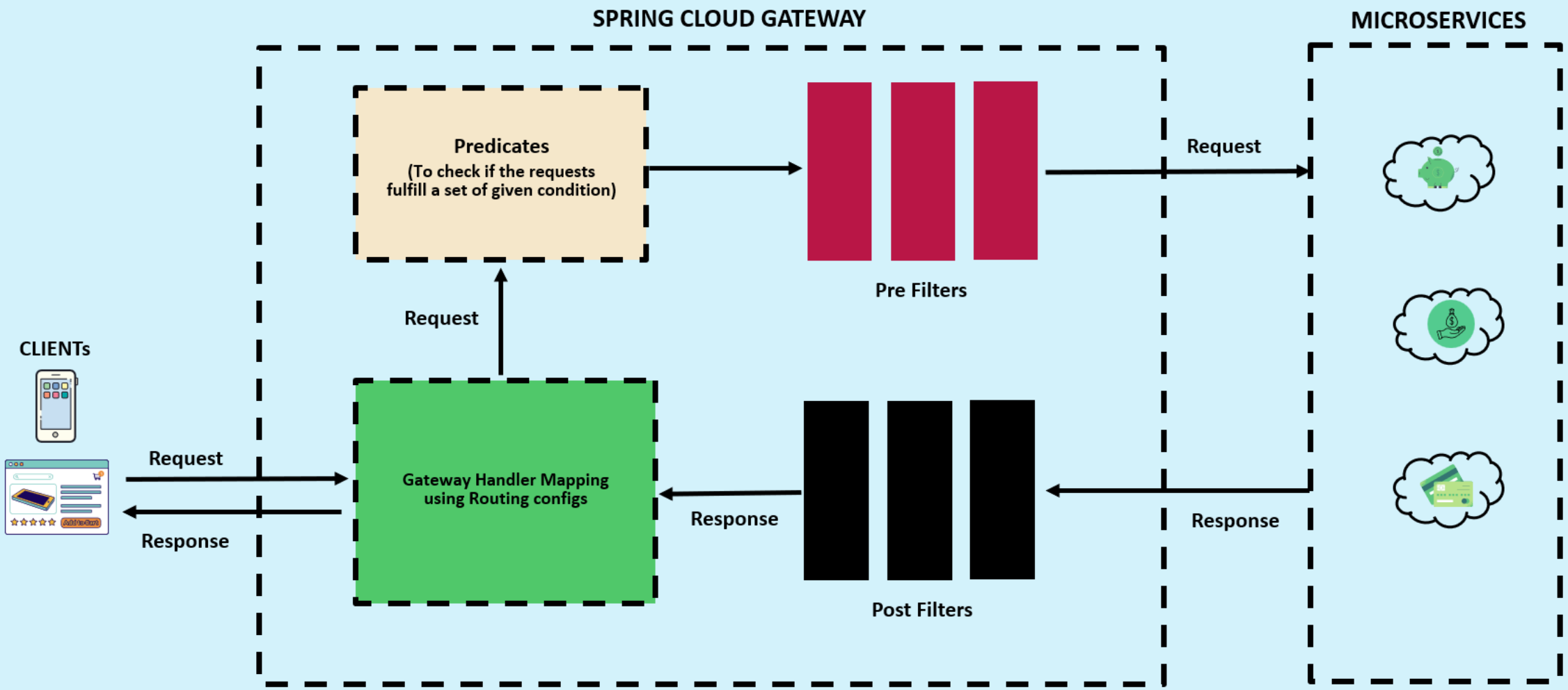
The service gateway sits between all calls from the client to the individual services & acts as a central Policy Enforcement Point (PEP) like below,

- Routing (Both Static & Dynamic)
- Security (Authentication & Authorization)
- Logging, Auditing and Metrics collection

Spring Cloud Gateway is the preferred API gateway compared to zuul. Because Spring Cloud Gateway built on Spring Reactor & Spring WebFlux, provides circuit breaker integration, service discovery with Eureka, non-blocking in nature, has a superior performance compared to that of Zuul.

Spring Cloud Gateway Internal Architecture

eazy bytes



When the client makes a request to the Spring Cloud Gateway, the Gateway Handler Mapping first checks if the request matches a route. This matching is done using the predicates. If it matches the predicate then the request is sent to the pre filters followed by actual microservices. The response will travel through post filters.

Steps to create Spring Cloud Gateway

eazy bytes

Below are the steps to make a microservice application to register and act as a Eureka client,

1. **Set up a new Spring Boot project:** Start by creating a new Spring Boot project using your preferred IDE or by using Spring Initializr (https://start.spring.io/). Include the spring-cloud-starter-gateway, spring-cloud-starter-config & spring-cloud-starter-netflix-eureka-client maven dependencies.

2. **Configure the properties:** In the application properties or YAML file, add the following configurations. Make routing configurations using RouteLocatorBuilder

```
eureka:
 instance:
  preferIpAddress: true
 client:
  registerWithEureka: true
  fetchRegistry: true
  serviceUrl:
   defaultZone: http://localhost:8070/eureka/
spring:
 cloud:
  gateway:
   discovery:
    locator:
     enabled: true
     lowerCaseServiceId: true
```

Steps to create Spring Cloud Gateway

eazy bytes

3 **Configure the routing config:** Make routing configurations using RouteLocatorBuilder like shown below,

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

4 **Build and run the application:** Build your project and run it as a Spring Boot application. Invokes the APIs using http://localhost:8072 which is the gateway path.

RESILIENCY IN MICROSERVICES

eazy bytes

HOW DO WE AVOID CASCADING FAILURES?

One failed or slow service should not have a ripple effect on the other microservices. Like in the scenarios of multiple microservices are communicating, we need to make sure that the entire chain of microservices does not fail with the failure of a single microservice

HOW DO WE HANDLE FAILURES GRACEFULLY WITH FALLBACKS?

In a chain of multiple microservices, how do we build a fallback mechanism if one of the microservice is not working. Like returning a default value or return values from cache or call another service/DB to fetch the results etc.

HOW TO MAKE OUR SERVICES SELF-HEALING CAPABLE

In the cases of slow performing services, how do we configure timeouts, retries and give time for a failed services to recover itself.

Ensuring system stability and resilience is crucial for providing a reliable service to users. One of the critical aspects in achieving a stable and resilient system for production is managing the integration points between services over a network.

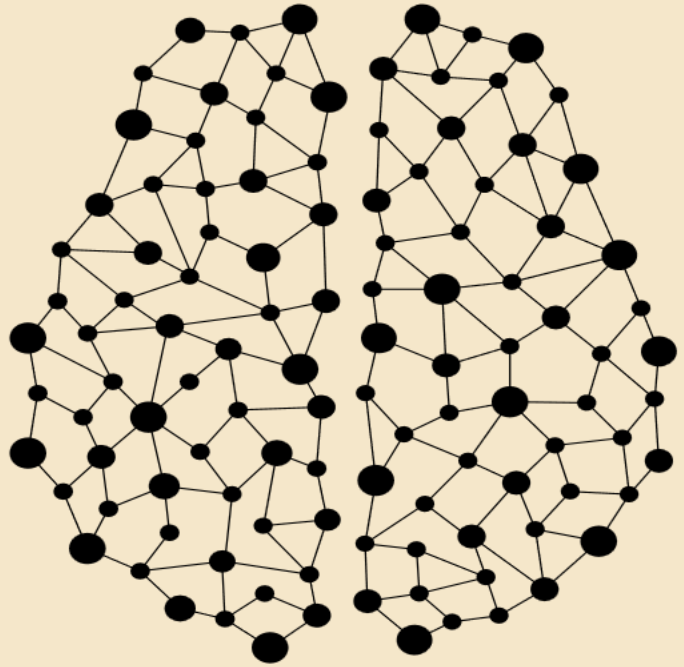
There exist various patterns for building resilient applications. In the Java ecosystem, Hystrix, a library developed by Netflix, was widely used for implementing such patterns. However, Hystrix entered maintenance mode in 2018 and is no longer being actively developed. To address this, Resilience4J has gained significant popularity, stepping in to fill the gap left by Hystrix. Resilience4J provides a comprehensive set of features for building resilient applications and has become a go-to choice for Java developers.

RESILIENCY USING RESILIENCE4J

eazy bytes

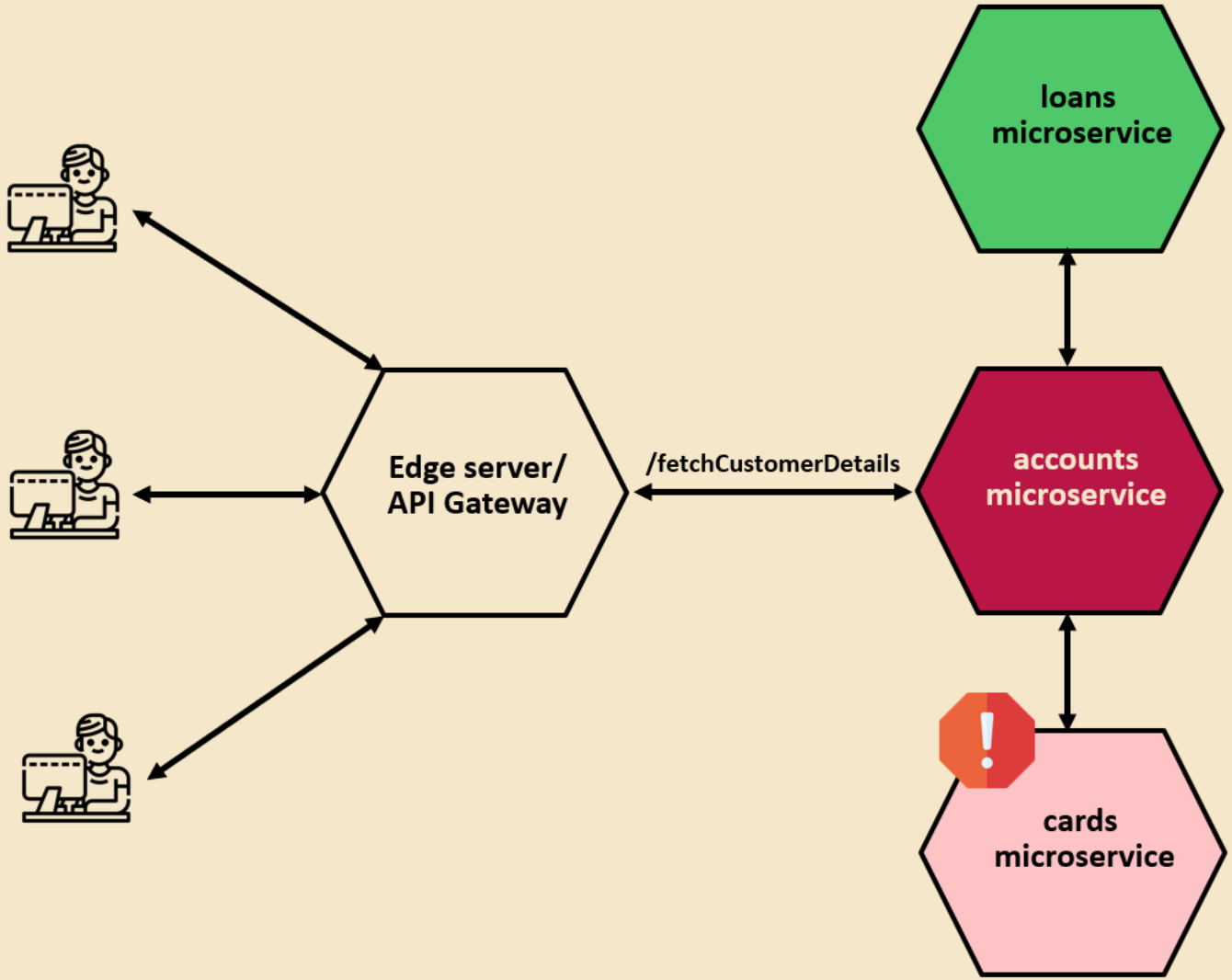
Resilience4j is a lightweight fault tolerance library designed for functional programming. It offers the following patterns for increasing fault tolerance due to network problems or failure of any of the multiple services:

- **Circuit breaker** - Used to stop making requests when a service invoked is failing
- **Fallback** - Alternative paths to failing requests
- **Retry** - Used to make retries when a service has temporarily failed
- **Rate limit** - Limits the number of calls that a service receives in a time
- **Bulkhead** - Limits the number of outgoing concurrent requests to a service to avoid overloading



TYPICAL SCENARIO IN MICROSERVICES

eazy bytes

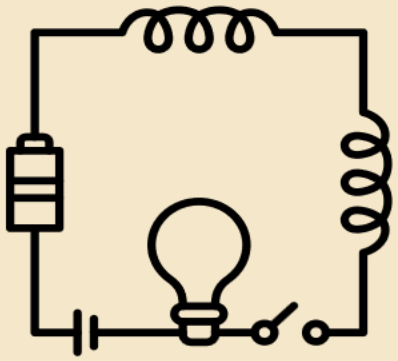


When a microservice responds slowly or fails to function, it can lead to the depletion of resource threads on the Edge server and intermediate services. This, in turn, has a negative impact on the overall performance of the microservice network.

To handle this kind of scenarios, we can use Circuit Breaker pattern

CIRCUIT BREAKER PATTERN

eazy bytes



In an electrical system, a circuit breaker is a safety device designed to protect the electrical circuit from excessive current, preventing damage to the circuit or potential fire hazards. It automatically interrupts the flow of electricity when it detects a fault, such as a short circuit or overload, to ensure the safety and stability of the system.

The Circuit Breaker pattern in software development takes its inspiration from the concept of an electrical circuit breaker found in electrical systems.

In a distributed environment, calls to remote resources and services can fail due to transient faults, such as slow network connections, timeouts, or the resources being overcommitted or temporarily unavailable. These faults typically correct themselves after a short period of time, and a robust cloud application should be prepared to handle them.

The Circuit Breaker pattern which inspired from electrical circuit breaker will monitor the remote calls. If the calls take too long, the circuit breaker will intercede and kill the call. Also, the circuit breaker will monitor all calls to a remote resource, and if enough calls fail, the circuit break implementation will pop, failing fast and preventing future calls to the failing remote resource.

The Circuit Breaker pattern also enables an application to detect whether the fault has been resolved. If the problem appears to have been fixed, the application can try to invoke the operation.

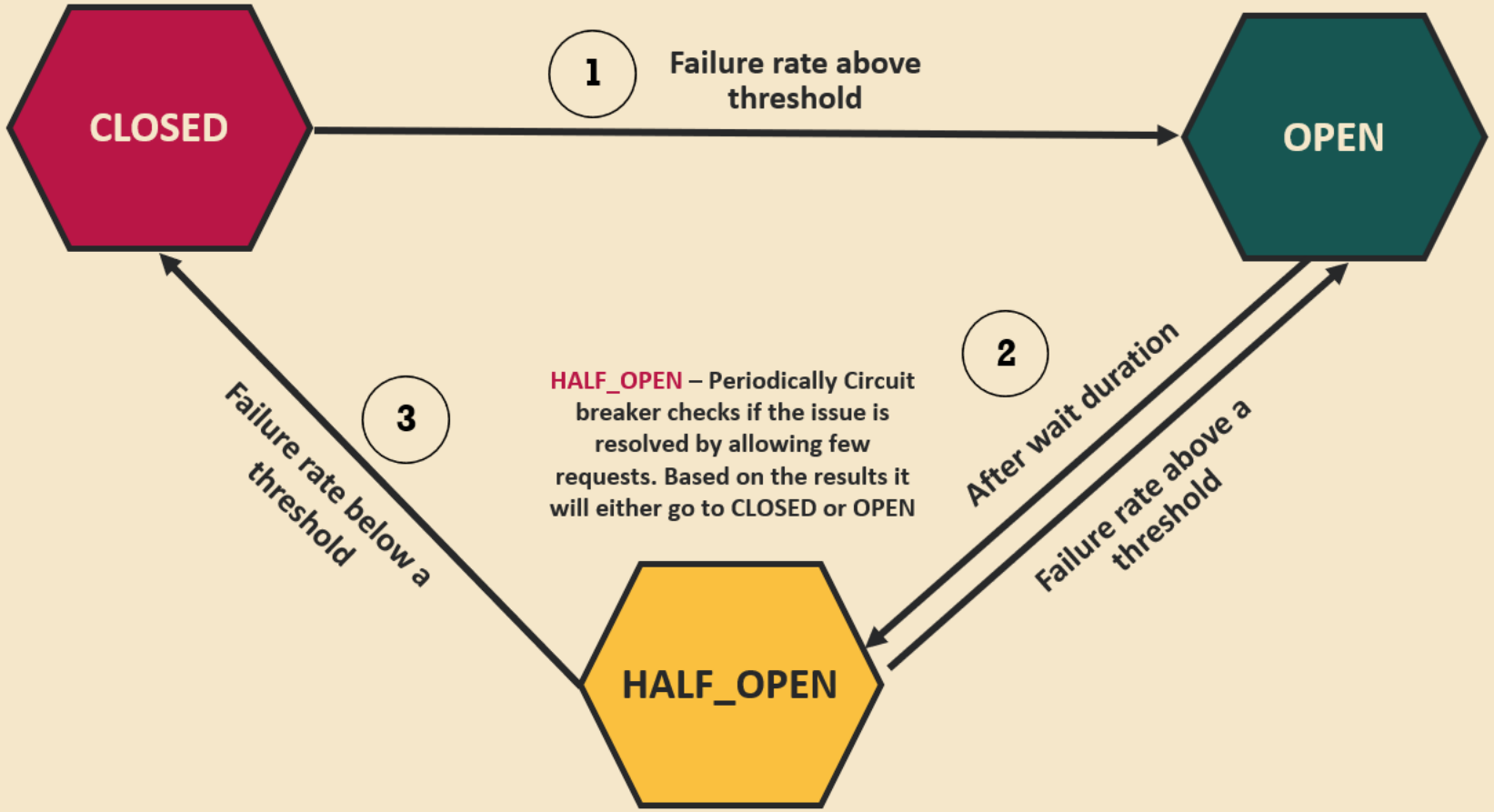
The advantages with circuit breaker pattern are,

- ✓ Fail fast
- ✓ Fail gracefully
- ✓ Recover seamlessly

CIRCUIT BREAKER PATTERN

eazy bytes

In Resilience4j, the circuit breaker is implemented via three states



CIRCUIT BREAKER PATTERN

eazy bytes

Below are the steps to build a circuit breaker pattern using Spring Cloud Gateway filter,

1. **Add maven dependency:** Add spring-cloud-starter-circuitbreaker-reactor-resilience4j maven dependency inside pom.xml

2. **Add circuit breaker filter:** Inside the method where we are creating a bean of RouteLocator, add a filter of circuit breaker like highlighted below and create a REST API handling the fallback uri `/contactSupport`

```
@Bean
public RouteLocator myRoutes(RouteLocatorBuilder builder) {
   return builder.routes()
               .route(p -> p.path("/eazybank/accounts/**")
                       .filters(f -> f.rewritePath("/eazybank/accounts/(?<segment>.*)","/${segment}")
                       .addResponseHeader("X-Response-Time",new Date().toString())
                       .circuitBreaker(config -> config.setName("accountsCircuitBreaker")
                       .setFallbackUri("forward:/contactSupport")))
                       .uri("lb://ACCOUNTS")).build();
}
```

3. **Add properties:** Add the below properties inside the application.yml file,

```
resilience4j.circuitbreaker:
  configs:
    default:
      slidingWindowSize: 10
      permittedNumberOfCallsInHalfOpenState: 2
      failureRateThreshold: 50
      waitDurationInOpenState: 10000
```

CIRCUIT BREAKER PATTERN

eazy bytes

Below are the steps to build a circuit breaker pattern using normal Spring Boot service,

1 **Add maven dependency:** Add spring-cloud-starter-circuitbreaker-resilience4j maven dependency inside pom.xml

2 **Add circuit breaker related changes in Feign Client interfaces like shown below:**

```
@FeignClient(name= "cards", fallback = CardsFallback.class)
public interface CardsFeignClient {

   @GetMapping(value = "/api/fetch",consumes = "application/json")
   public ResponseEntity<CardsDto> fetchCardDetails(@RequestHeader("eazybank-correlation-id")
                                    String correlationId, @RequestParam String mobileNumber);

}
```

```
@Component
public class CardsFallback implements CardsFeignClient{
   @Override
   public ResponseEntity<CardsDto> fetchCardDetails(String correlationId, String mobileNumber) {
      return null;
   }
}
```

CIRCUIT BREAKER PATTERN

eazy bytes

3 **Add properties:** Add the below properties inside the application.yml file,

```
spring:
 cloud:
  openfeign:
   circuitbreaker:
    enabled: true
resilience4j.circuitbreaker:
    configs:
      default:
        slidingWindowSize: 5
        failureRateThreshold: 50
        waitDurationInOpenState: 10000
        permittedNumberOfCallsInHalfOpenState: 2
```

RETRY PATTERN

eazy bytes

The retry pattern will make configured multiple retry attempts when a service has temporarily failed. This pattern is very helpful in the scenarios like network disruption where the client request may successful after a retry attempt.

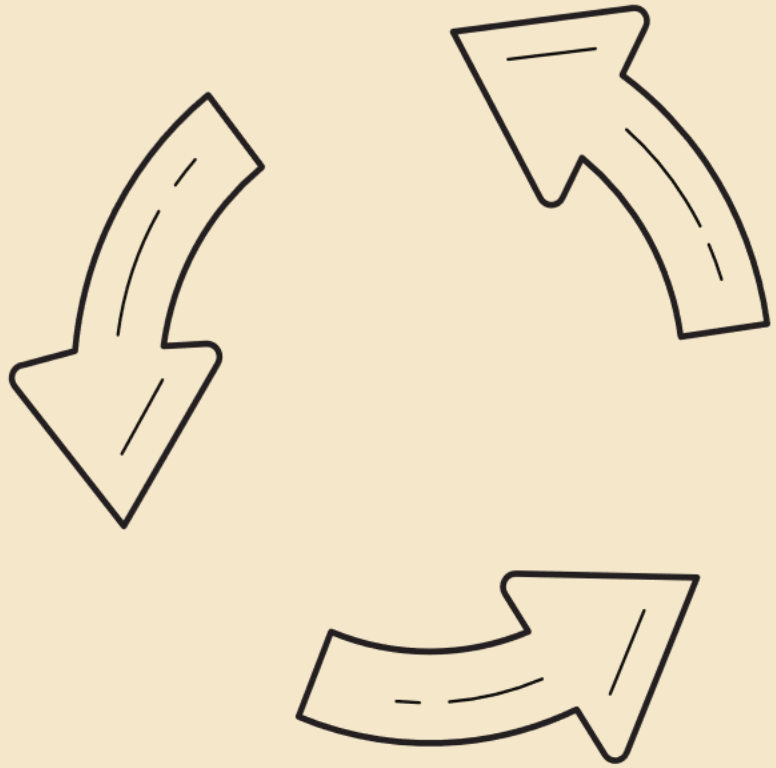
Here are some key components and considerations of implementing the Retry pattern in microservices:

- **Retry Logic:** Determine when and how many times to retry an operation. This can be based on factors such as error codes, exceptions, or response status.

- **Backoff Strategy:** Define a strategy for delaying retries to avoid overwhelming the system or exacerbating the underlying issue. This strategy can involve gradually increasing the delay between each retry, known as exponential backoff.

- **Circuit Breaker Integration:** Consider combining the Retry pattern with the Circuit Breaker pattern. If a certain number of retries fail consecutively, the circuit can be opened to prevent further attempts and preserve system resources.

- **Idempotent Operations:** Ensure that the retried operation is idempotent, meaning it produces the same result regardless of how many times it is invoked. This prevents unintended side effects or duplicate operations.



RETRY PATTERN

eazy bytes

Below are the steps to build a retry pattern using Spring Cloud Gateway filter,

1 **Add Retry filter:** Inside the method where we are creating a bean of RouteLocator, add a filter of retry like highlighted below,

```
@Bean
public RouteLocator myRoutes(RouteLocatorBuilder builder) {
  return builder.routes()
      .route(p -> p.path("/eazybank/loans/**")
          .filters(f -> f.rewritePath("/eazybank/loans/(?<segment>.*)","/${segment}")
              .addResponseHeader("X-Response-Time",new Date().toString())
              .retry(retryConfig -> retryConfig.setRetries(3).setMethods(HttpMethod.GET)
                  .setBackoff(Duration.ofMillis(100),Duration.ofMillis(1000),2,true)))
          .uri("lb://LOANS")).build();
}
```

RETRY PATTERN

eazy bytes

Below are the steps to build a retry pattern using normal Spring Boot service,

1 **Add Retry pattern annotations:** Choose a method and mention retry pattern related annotation along with the below configs. Post that create a fallback method matching the same method signature like we discussed inside the course,

```
@Retry(name = "getBuildInfo", fallbackMethod = "getBuildInfoFallBack")
@GetMapping("/build-info")
public ResponseEntity<String> getBuildInfo() {

}

private ResponseEntity<String> getBuildInfoFallBack(Throwable t) {

}
```

RETRY PATTERN

eazy bytes

2

Add properties: Add the below properties inside the application.yml file,

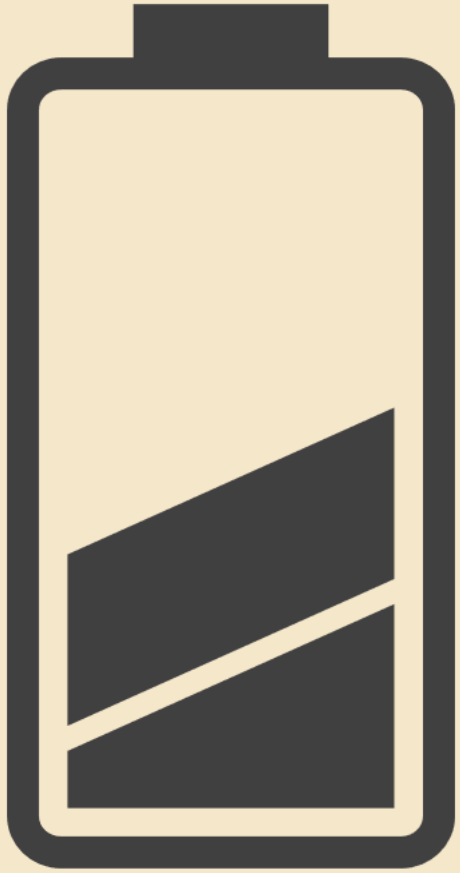
```
resilience4j.retry:
  configs:
    default:
      maxRetryAttempts: 3
      waitDuration: 500
      enableExponentialBackoff: true
      exponentialBackoffMultiplier: 2
      retryExceptions:
       - java.util.concurrent.TimeoutException
      ignoreExceptions:
       - java.lang.NullPointerException
```

RATE LIMITTER PATTERN

eazy bytes

The Rate Limiter pattern in microservices is a design pattern that helps control and limit the rate of incoming requests to a service or API. It is used to prevent abuse, protect system resources, and ensure fair usage of the service.

In a microservices architecture, multiple services may depend on each other and make requests to communicate. However, unrestricted and uncontrolled requests can lead to performance degradation, resource exhaustion, and potential denial-of-service (DoS) attacks. The Rate Limiter pattern provides a mechanism to enforce limits on the rate of incoming requests.



Implementing the Rate Limiter pattern helps protect microservices from being overwhelmed by excessive or malicious requests. It ensures the stability, performance, and availability of the system while providing controlled access to resources. By enforcing rate limits, the Rate Limiter pattern helps maintain a fair and reliable environment for both the service provider and its consumers.

When a user surpasses the permitted number of requests within a designated time frame, any additional requests are declined with an HTTP 429 - Too Many Requests status. The specific limit is enforced based on a chosen strategy, such as limiting requests per session, IP address, user, or tenant. The primary objective is to maintain system availability for all users, especially during challenging circumstances. This exemplifies the essence of resilience. Additionally, the Rate Limiter pattern proves beneficial for providing services to users based on their subscription tiers. For instance, distinct rate limits can be defined for basic, premium, and enterprise users.

RATE LIMITTER PATTERN

eazy bytes

Below are the steps to build a rate limitter pattern using Spring Cloud Gateway filter,

1. **Add maven dependency:** Add spring-boot-starter-data-redis-reactive maven dependency inside pom.xml and make sure a redis container started. Mention redis connection details inside the application.yml file

2. **Add rate limitter filter:** Inside the method where we are creating a bean of RouteLocator, add a filter of rate limitter like highlighted below and creating supporting beans of RedisRateLimiter and KeyResolver

```
@Bean
public RouteLocator myRoutes(RouteLocatorBuilder builder) {
    return builder.routes()
            .route(p -> p.path("/eazybank/cards/**")
                    .filters(f -> f.rewritePath("/eazybank/cards/(?<segment>.*)","/${segment}")
                       .addResponseHeader("X-Response-Time",new Date().toString())
                       .requestRateLimiter(config ->
                         config.setRateLimiter(redisRateLimiter()).setKeyResolver(userKeyResolver())))
                  .uri("lb://CARDS")).build();
}

@Bean
public RedisRateLimiter redisRateLimiter() {
    return new RedisRateLimiter(1, 1, 1);
}

@Bean
KeyResolver userKeyResolver() {
    return exchange -> Mono.justOrEmpty(exchange.getRequest().getHeaders().getFirst("user"))
            .defaultIfEmpty("anonymous");
}
```

RATE LIMITTER PATTERN

eazy bytes

Below are the steps to build a rate limiter pattern using normal Spring Boot service,

1 **Add Rate limitter pattern annotations:** Choose a method and mention rate limitter pattern related annotation along with the below configs. Post that create a fallback method matching the same method signature like we discussed inside the course,

```
@RateLimiter(name = "getJavaVersion", fallbackMethod = "getJavaVersionFallback")
@GetMapping("/java-version")
public ResponseEntity<String> getJavaVersion() {

}

private ResponseEntity<String> getJavaVersionFallback(Throwable t) {

}
```

2 **Add properties:** Add the below properties inside the application.yml file,

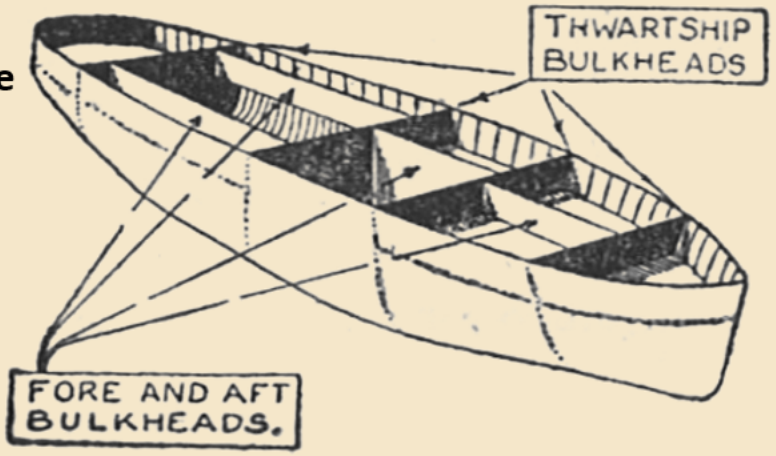
```
resilience4j.ratelimiter:
  configs:
    default:
      timeoutDuration: 5000
      limitRefreshPeriod: 5000
      limitForPeriod: 1
```

BULKHEAD PATTERN

eazy bytes

The Bulkhead pattern in software architecture is a design pattern that aims to improve the resilience and isolation of components or services within a system. It draws inspiration from the concept of bulkheads in ships, which are physical partitions that prevent the flooding of one compartment from affecting others, enhancing the overall stability and safety of the vessel.

In the context of software systems, the Bulkhead pattern is used to isolate and limit the impact of failures or high loads in one component from spreading to other components. It helps ensure that a failure or heavy load in one part of the system does not bring down the entire system, enabling other components to continue functioning independently.

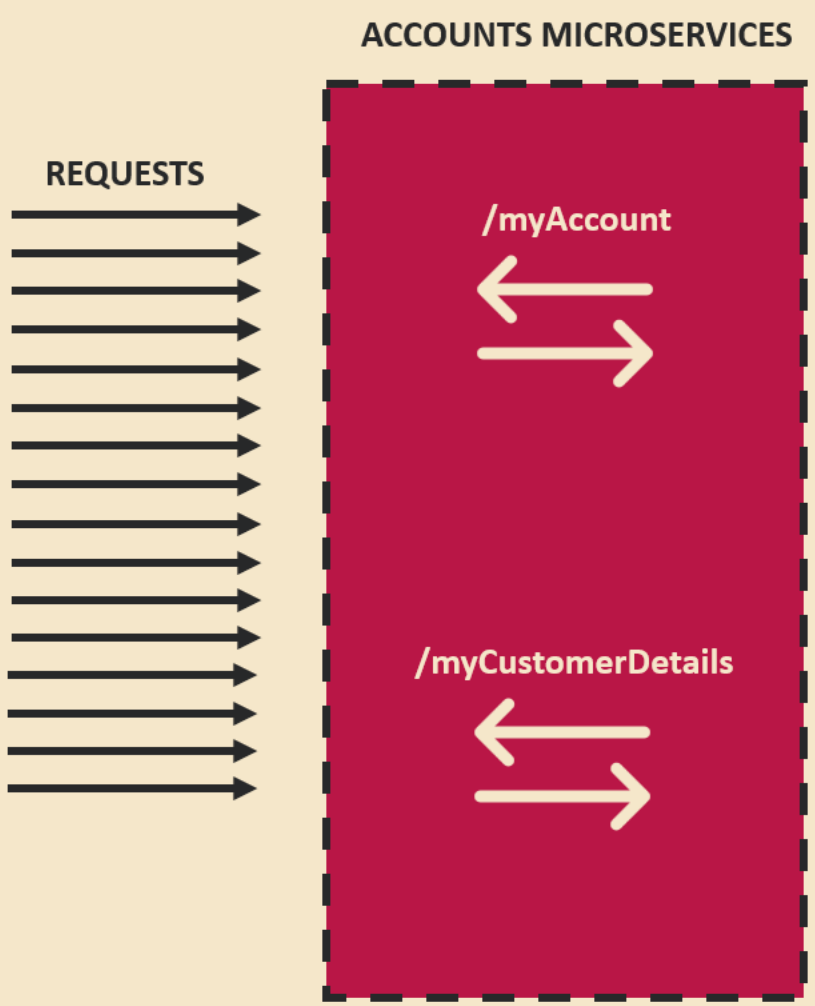


Bulkhead Pattern helps us to allocate limit the resources which can be used for specific services. So that resource exhaustion can be reduced.

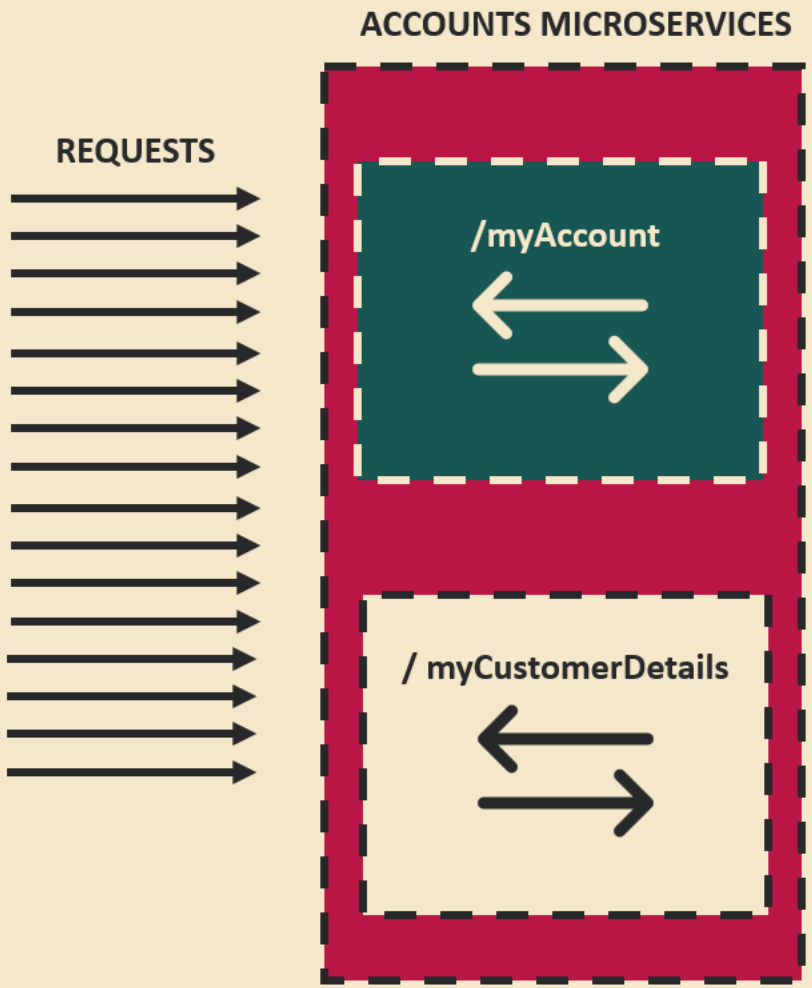
The Bulkhead pattern is particularly useful in systems that require high availability, fault tolerance, and isolation between components. By compartmentalizing components and enforcing resource boundaries, the Bulkhead pattern enhances the resilience and stability of the system, ensuring that failures or heavy loads in one area do not bring down the entire system.

BULKHEAD PATTERN

eazy bytes



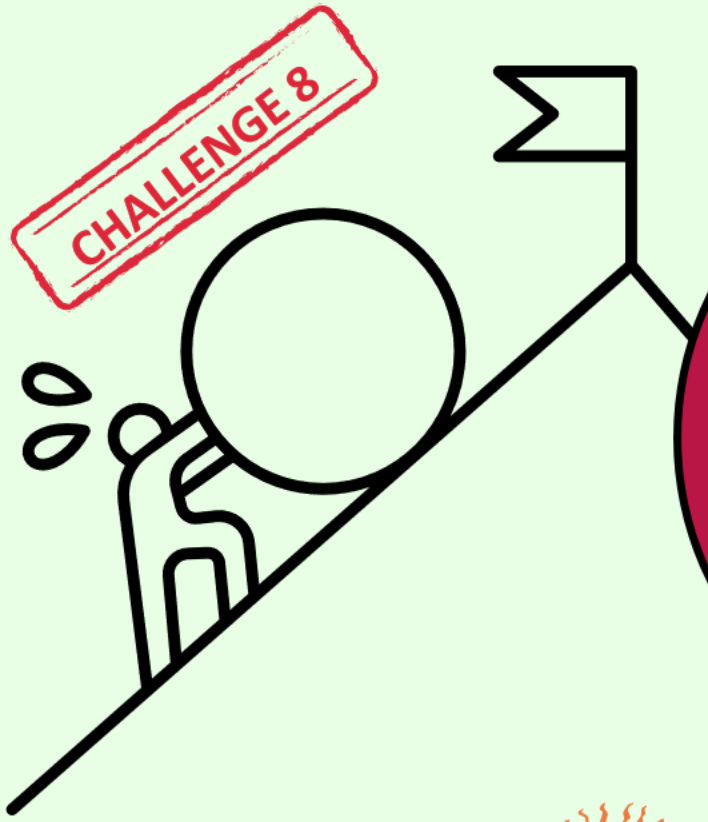
Without Bulkhead, /myCustomerDetails will start eating all the threads, resources available which will effect the performance of /myAccount



With Bulkhead, /myCustomerDetails and /myAccount will have their own resources, threads pool defined

OBSERVABILITY AND MONITORING OF MICROSERVICES

eazy bytes



DEBUGGING A PROBLEM IN MICROSERVICES ?

How do we trace transactions across multiple services, containers and try to find where exactly the problem or bug is?

How do we combine all the logs from multiple services into a central location where they can be indexed, searched, filtered, and grouped to find bugs that are contributing to a problem?

MONITORING PERFORMANCE OF SERVICE CALLS?

How can we track the path of a specific chain service call through our microservices network, and see how long it took to complete at each microservice?

MONITORING SERVICES METRICS & HEALTH ?

How can we easily and efficiently monitor the metrics like CPU usage, JVM metrics, etc. for all the microservices applications in our network?

How can we monitor the status and health of all of our microservices applications in a single place, and create alerts and notifications for any abnormal behavior of the services?

Observability and **monitoring** solve the challenge of identifying and resolving above problems in microservices architectures before they cause outages.

WHAT IS OBSERVABILITY ?

eazy bytes

Observability is the ability to understand the internal state of a system by observing its outputs. In the context of microservices, observability is achieved by collecting and analyzing data from a variety of sources, such as metrics, logs, and traces.

The three pillars of observability are:

Metrics: Metrics are quantitative measurements of the health of a system. They can be used to track things like CPU usage, memory usage, and response times.

Logs: Logs are a record of events that occur in a system. They can be used to track things like errors, exceptions, and other unexpected events.

Traces: Traces are a record of the path that a request takes through a system. They can be used to track the performance of a request and to identify bottlenecks.

By collecting and analyzing data from these three sources, you can gain a comprehensive understanding of the internal state of your microservices architecture. This understanding can be used to identify and troubleshoot problems, improve performance, and ensure the overall health of your system.

WHAT IS MONITORING ?

eazy bytes

Monitoring in microservices involves checking the telemetry data available for the application and defining alerts for known failure states. This process collects and analyzes data from a system to identify and troubleshoot problems, as well as track the health of individual microservices and the overall health of the microservices network.

Monitoring in microservices is important because it allows you to:

- **Identify and troubleshoot problems:** By collecting and analyzing data from your microservices, you can identify problems before they cause outages or other disruptions.

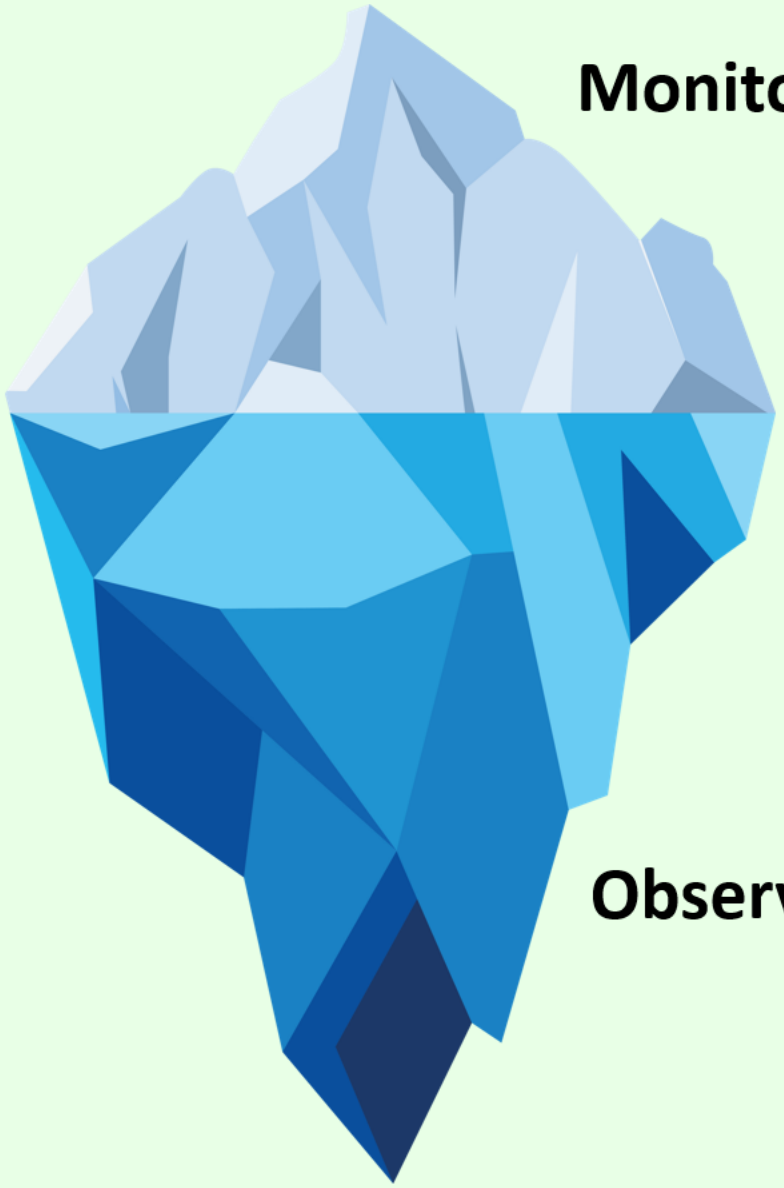
- Track the health of your microservices: **Monitoring can help you to track the health of your microservices, so you can identify any microservices that are underperforming or that are experiencing problems.**

- **Optimize your microservices:** By monitoring your microservices, you can identify areas where you can optimize your microservices to improve performance and reliability.

Monitoring and observability can be considered as two sides of the same coin. Both rely on the same types of telemetry data to enable insight into software distributed systems. Those data types — metrics, traces, and logs — are often referred to as the three pillars of observability.

Observability vs. Monitoring

eazy bytes



Feature	Monitoring	Observability
Purpose	Identify and troubleshoot problems	Understand the internal state of a system
Data	Metrics, traces, and logs	Metrics, traces, logs, and other data sources
Goal	Identify problems	Understand how a system works
Approach	Reactive	Proactive

In other words, monitoring is about collecting data and observability is about understanding data.

Monitory is reacting to problems while observavility is fixing them in real time.

Logging

eazy bytes

Logs are discrete records of events that happen in software applications over time. They contain a timestamp that indicates when the event happened, as well as information about the event and its context. This information can be used to answer questions like "What happened at this time?", "Which thread was processing the event?", or "Which user/tenant was in the context?"

Logs are essential tools for troubleshooting and debugging tasks. They can be used to reconstruct what happened at a specific point in time in a single application instance. Logs are typically categorized according to the type or severity of the event, such as trace, debug, info, warn, and error. This allows us to log only the most severe events in production, while still giving us the chance to change the log level temporarily during debugging.

Logging in Monolithic Apps

In monolithic apps, all of the code is in a single codebase. This means that all of the logs are also in a single location. This makes it easy to find and troubleshoot problems, as you only need to look in one place.

Logging in Microservices

Logging in microservices is complex. This is because each service has its own logs. This means that you need to look in multiple places to find all of the logs for a particular request.

To address this challenge, microservices architectures often use centralized logging. Centralized logging collects logs from all of the services in the architecture and stores them in a single location. This makes it easier to find and troubleshoot problems, as you only need to look in one place

Managing logs with Grafana, Loki & Promtail

eazy bytes

Grafana is an open-source analytics and interactive visualization web application. It provides charts, graphs, and alerts for the web when connected to supported data sources. It can be easily installed using Docker or Docker Compose.

Grafana is a popular tool for visualizing metrics, logs, and traces from a variety of sources. It is used by organizations of all sizes to monitor their applications and infrastructure.

Grafana Loki is a horizontally scalable, highly available, and cost-effective log aggregation system. It is designed to be easy to use and to scale to meet the needs of even the most demanding applications.

Promtail is a lightweight log agent that ships logs from your containers to Loki. It is easy to configure and can be used to collect logs from a wide variety of sources.

Together, Grafana Loki and Promtail provide a powerful logging solution that can help you to understand and troubleshoot your applications.

Grafana provides visualization of the log lines captured within Loki.

FROM GRAFANA LOKI 3.0,
Alloy replaced Promtail

Managing logs with Grafana, Loki & Alloy

eazy bytes

Grafana is an open-source analytics and interactive visualization web application. It provides charts, graphs, and alerts for the web when connected to supported data sources. It can be easily installed using Docker or Docker Compose.

Grafana is a popular tool for visualizing metrics, logs, and traces from a variety of sources. It is used by organizations of all sizes to monitor their applications and infrastructure.

Grafana Loki is a horizontally scalable, highly available, and cost-effective log aggregation system. It is designed to be easy to use and to scale to meet the needs of even the most demanding applications.

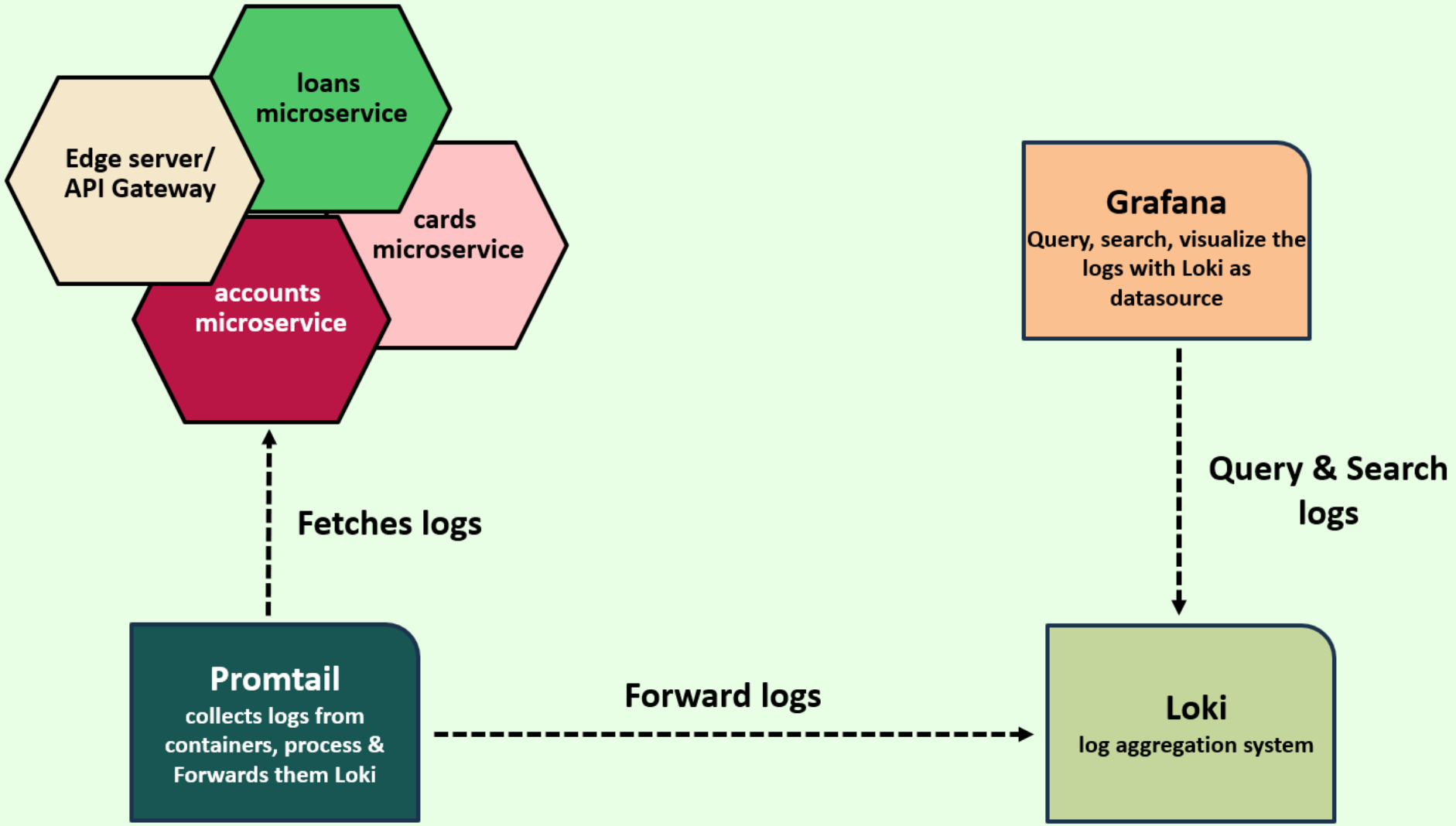
Grafana Alloy is a lightweight log agent that ships logs from your containers to Loki. It is easy to configure and can be used to collect logs from a wide variety of sources.

Together, Grafana Loki and Alloy provide a powerful logging solution that can help you to understand and troubleshoot your applications.

Grafana provides visualization of the log lines captured within Loki.

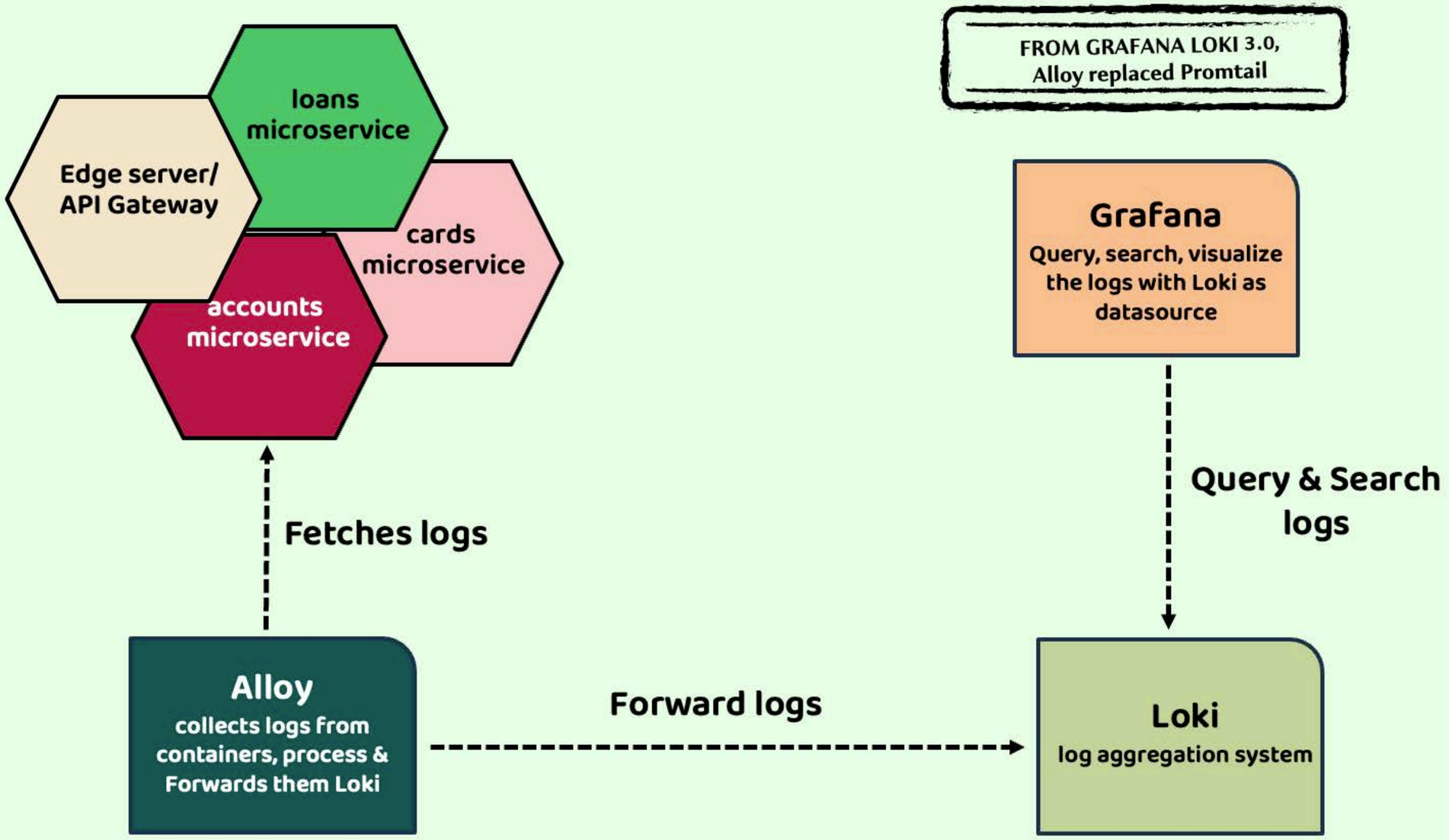
Managing logs with Grafana, Loki & Promtail

eazy bytes



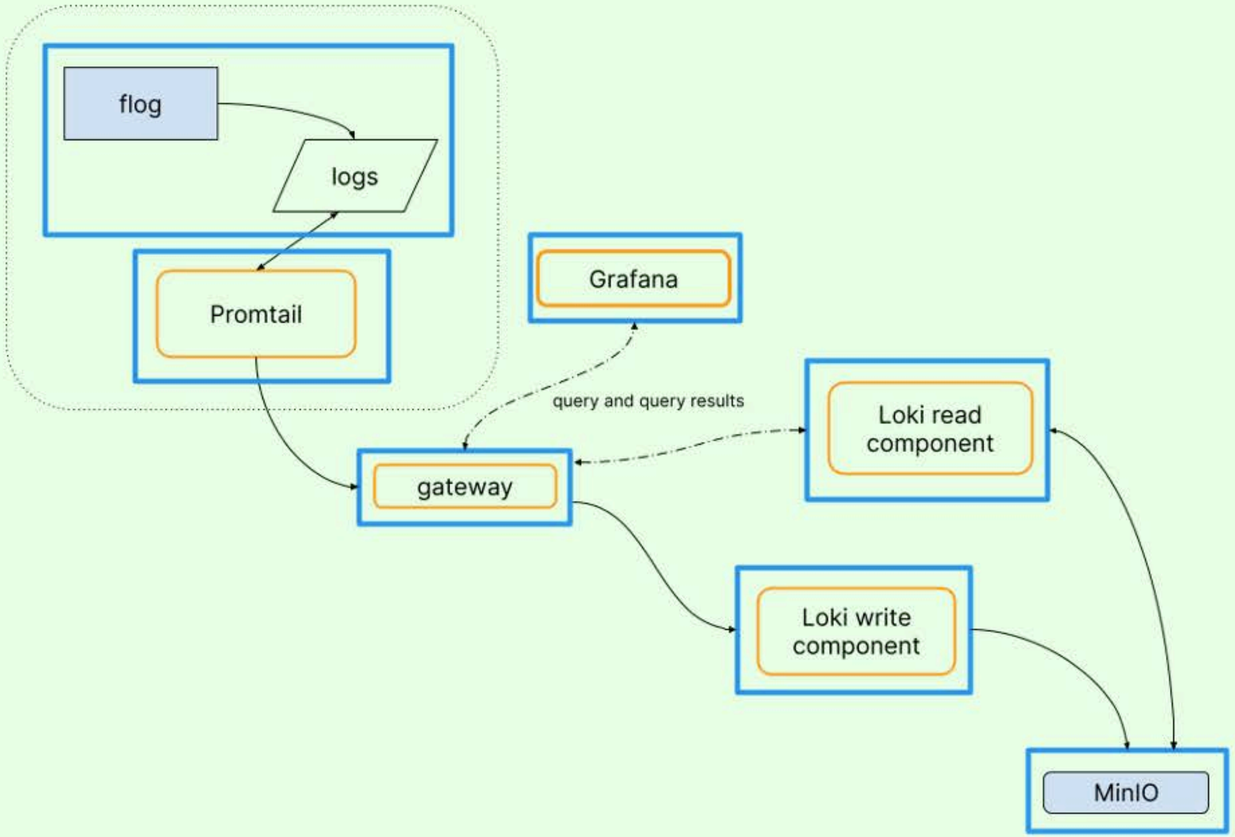
Managing logs with Grafana, Loki & Alloy

eazy bytes



Sample demo of logging using Grafana, Loki & promotail

eazy bytes



cloud-native applications generate logs as events and send them to the standard output, without being concerned about the processing or storage of those logs.

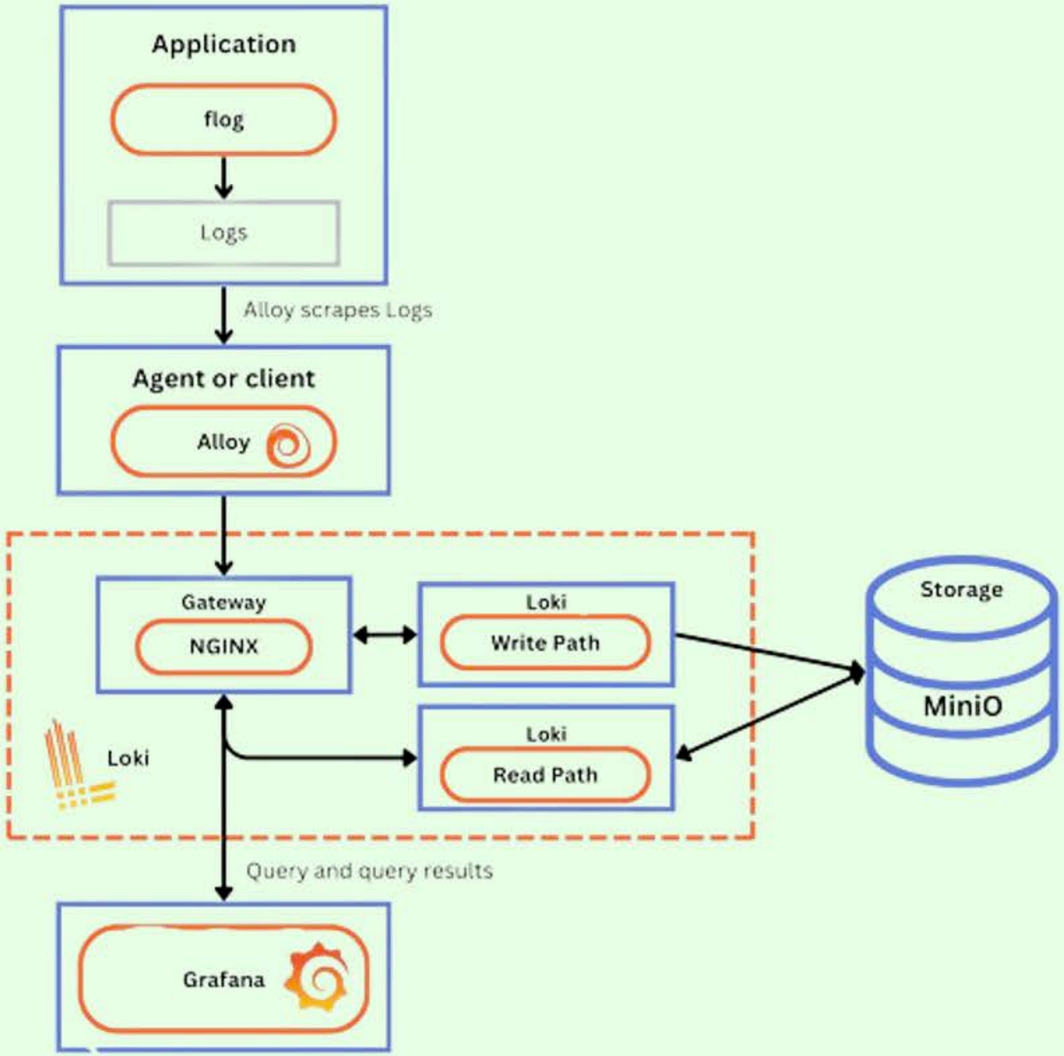
One advantage of treating logs as event streams and emitting them to stdout is that it decouples the application from the log processing infrastructure. The application can focus on its core functionality without being tied to a specific logging implementation or storage solution. The infrastructure, on the other hand, can handle the collection, aggregation, and storage of logs using appropriate tools and services.

15-Factor methodology recommends the same to treat logs as events streamed to the standard output and not concern with how they are processed or stored.

Sample demo of logging using Grafana, Loki & Alloy

eazy bytes

Reference: https://grafana.com/docs/loki/latest/get-started/quick-start/



FROM GRAFANA LOKI 3.0,
Alloy replaced Promtail

cloud-native applications generate logs as events and send them to the standard output, without being concerned about the processing or storage of those logs.

One advantage of treating logs as event streams and emitting them to stdout is that it decouples the application from the log processing infrastructure. The application can focus on its core functionality without being tied to a specific logging implementation or storage solution. The infrastructure, on the other hand, can handle the collection, aggregation, and storage of logs using appropriate tools and services.

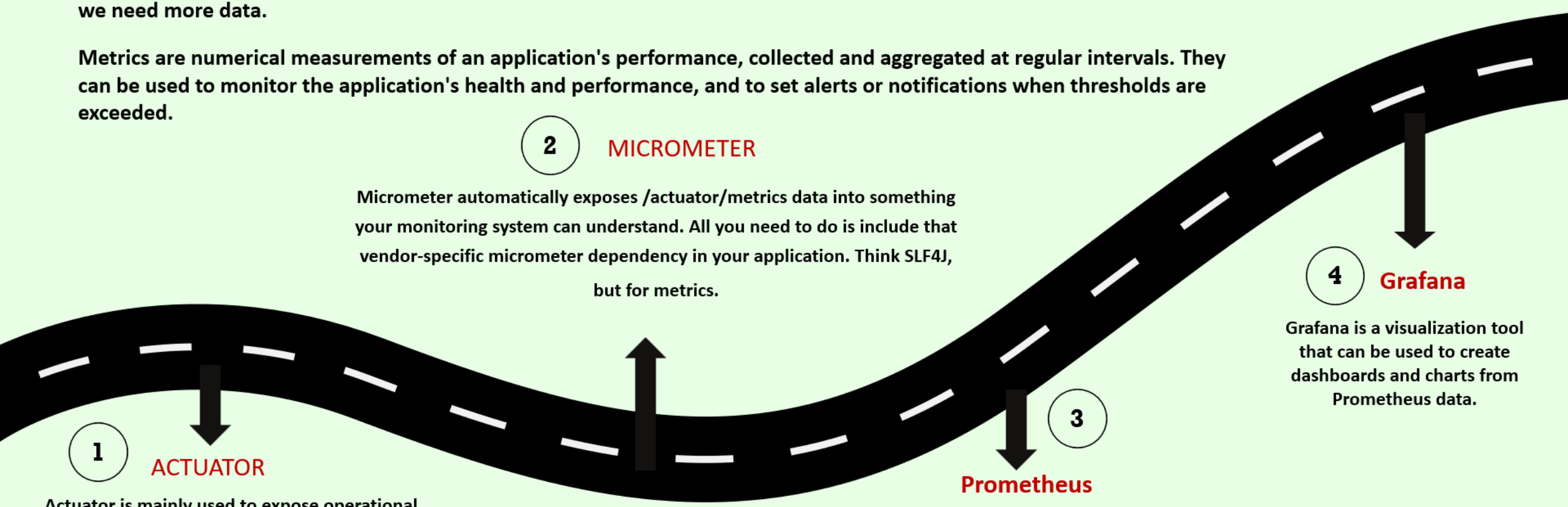
15-Factor methodology recommends the same to treat logs as events streamed to the standard output and not concern with how they are processed or stored.

Metrics & monitoring with Spring Boot Actuator, Micrometer, Prometheus & Grafana

eazy bytes

Event logs are essential for monitoring applications, but they don't provide enough data to answer all of the questions we need to know. To answer questions like CPU usage, memory usage, threads usage, error requests etc. & properly monitor, manage, and troubleshoot an application in production, we need more data.

Metrics are numerical measurements of an application's performance, collected and aggregated at regular intervals. They can be used to monitor the application's health and performance, and to set alerts or notifications when thresholds are exceeded.



2 MICROMETER

Micrometer automatically exposes /actuator/metrics data into something your monitoring system can understand. All you need to do is include that vendor-specific micrometer dependency in your application. Think SLF4J, but for metrics.

4 Grafana

Grafana is a visualization tool that can be used to create dashboards and charts from Prometheus data.

1 ACTUATOR

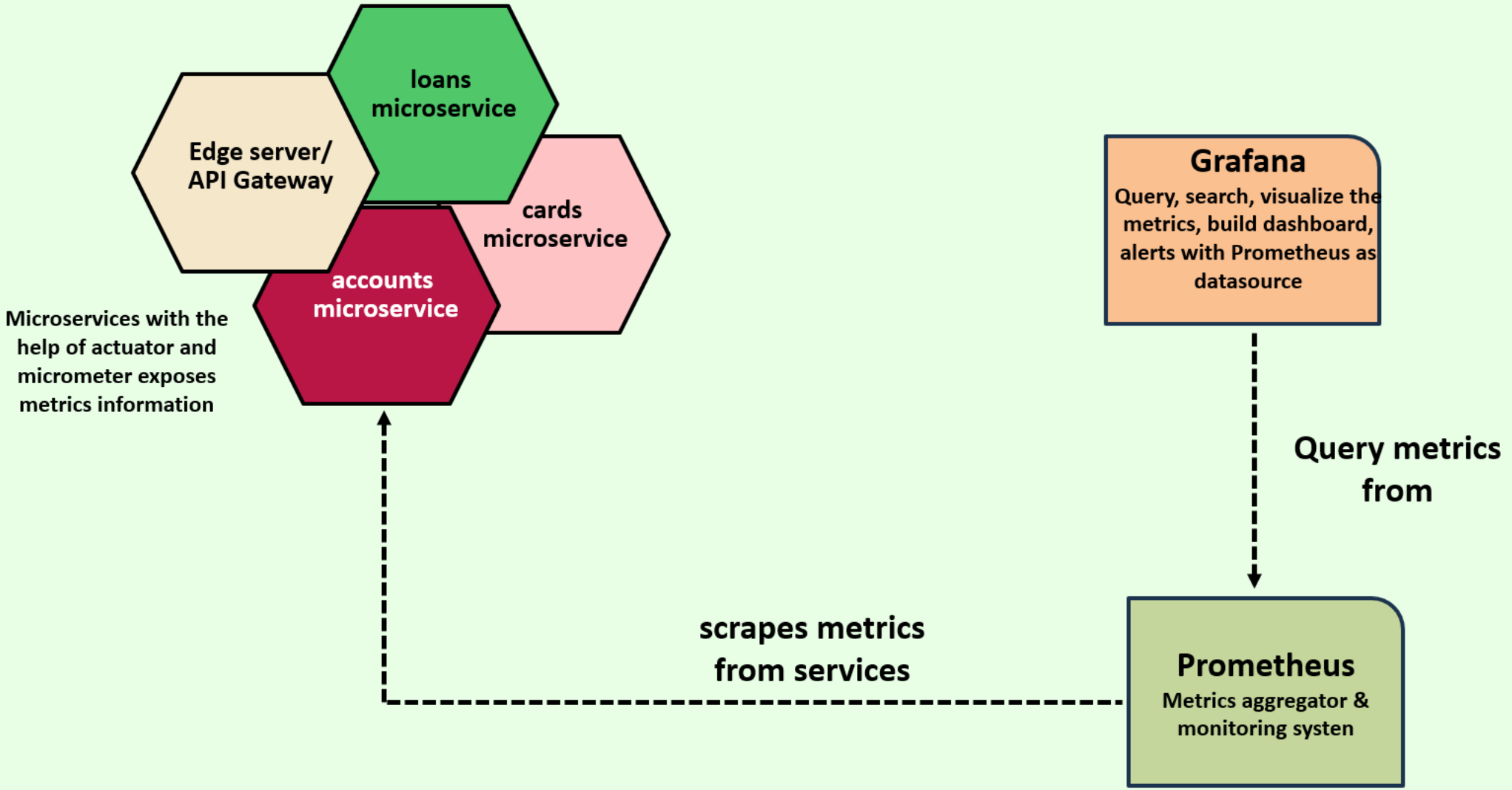
Actuator is mainly used to expose operational information about the running application — health, metrics, info, dump, env, etc. It uses HTTP endpoints or JMX beans to enable us to interact with it.

3 Prometheus

The most common format for exporting metrics is the one used by Prometheus, which is “an open-source systems monitoring and alerting toolkit”. Just as Loki aggregates and stores event logs, Prometheus does the same with metrics.

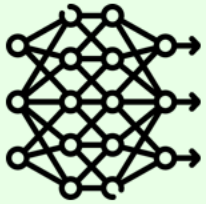
Metrics & monitoring with Spring Boot Actuator, Micrometer, Prometheus & Grafana

eazy bytes



Distributed tracing in microservices

eazy bytes



Event logs, health probes, and metrics offer a wealth of valuable information for deducing the internal condition of an application. Nevertheless, these sources fail to account for the distributed nature of cloud-native applications. Given that a user request often traverses multiple applications, we currently lack the means to effectively correlate data across application boundaries.

Distributed tracing is a technique used in microservices or cloud-native applications to understand and analyze the flow of requests as they propagate across multiple services and components. It helps in gaining insights into how requests are processed, identifying performance bottlenecks, and diagnosing issues in complex, distributed systems.

One possible solution to address this issue is to implement a straightforward approach where a unique identifier, known as a correlation ID, is generated for each request at the entry point of the system. This correlation ID can then be utilized in event logs and passed along to other relevant services involved in processing the request. By leveraging this correlation ID, we can retrieve all log messages associated with a specific transaction from multiple applications.

Distributed tracing encompasses three primary concepts:

Tags serve as metadata that offer supplementary details about the span context, including the request URI, the username of the authenticated user, or the identifier for a specific tenant.

A trace denotes the collection of actions tied to a request or transaction, distinguished by a **trace ID**. It consists of multiple spans that span across various services.

A span represents each individual stage of request processing, encompassing start and end timestamps, and is uniquely identified by the combination of trace ID and span ID.

Distributed tracing in microservices

eazy bytes

When a client request received at the edge server or the first service inside the network, a trace ID like 29cdbe2e21bc will be generated and it is going to be same throught the request

[gatewayserver, 29cdbe2e21bc, 259d5208]

Edge server/ API Gateway

accounts microservice

loans microservice

[loans, 29cdbe2e21bc, 9e5b7f81] – logger statement 1

[loans, 29cdbe2e21bc, 9e5b7f81] – logger statement 2

A Span ID is going to be generated which is specific to a service and the same will be present in all the logs of the service in a request

[accounts, 29cdbe2e21bc, 61fd3a92]

Apart from Trace ID and Span ID, a metadata information can also be attached to the logs using Tags. For example, here we added serive name which is accounts

cards microservice

[cards, 29cdbe2e21bc, 8f4b7a76] – logger statement 1

[cards, 29cdbe2e21bc, 8f4b7a76] – logger statement 2

Distributed tracing with OpenTelemetry, Tempo & Grafana

eazy bytes

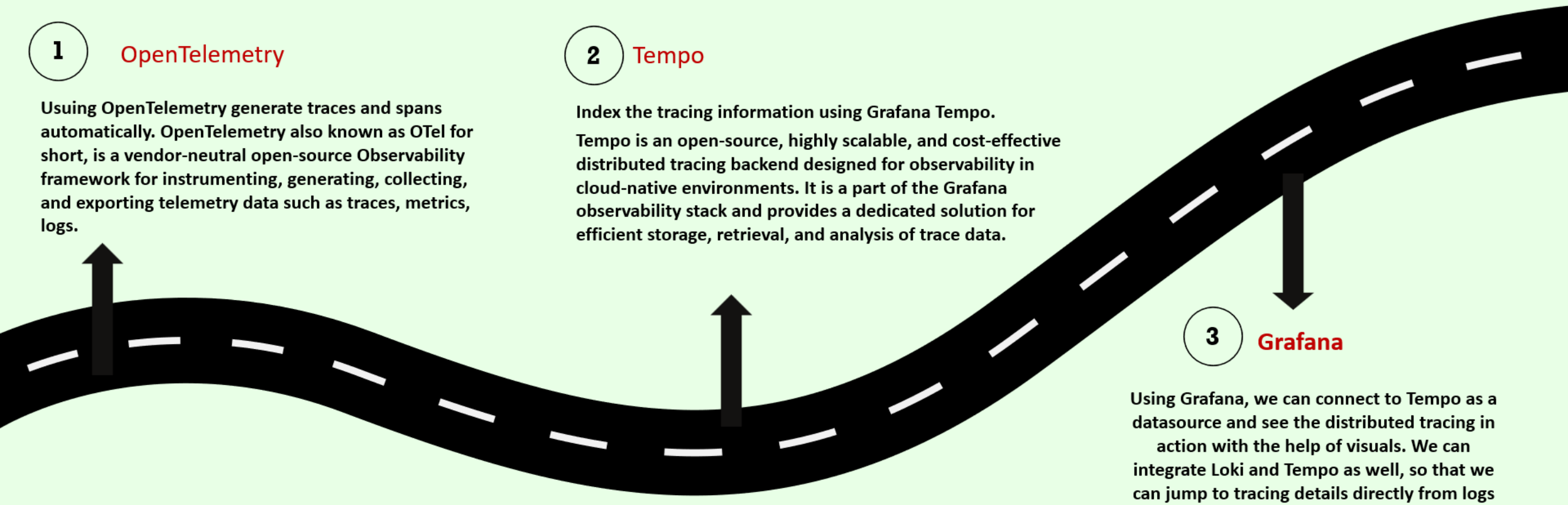
1 OpenTelemetry

Usuing OpenTelemetry generate traces and spans automatically. OpenTelemetry also known as OTel for short, is a vendor-neutral open-source Observability framework for instrumenting, generating, collecting, and exporting telemetry data such as traces, metrics, logs.

2 Tempo

Index the tracing information using Grafana Tempo.

Tempo is an open-source, highly scalable, and cost-effective distributed tracing backend designed for observability in cloud-native environments. It is a part of the Grafana observability stack and provides a dedicated solution for efficient storage, retrieval, and analysis of trace data.

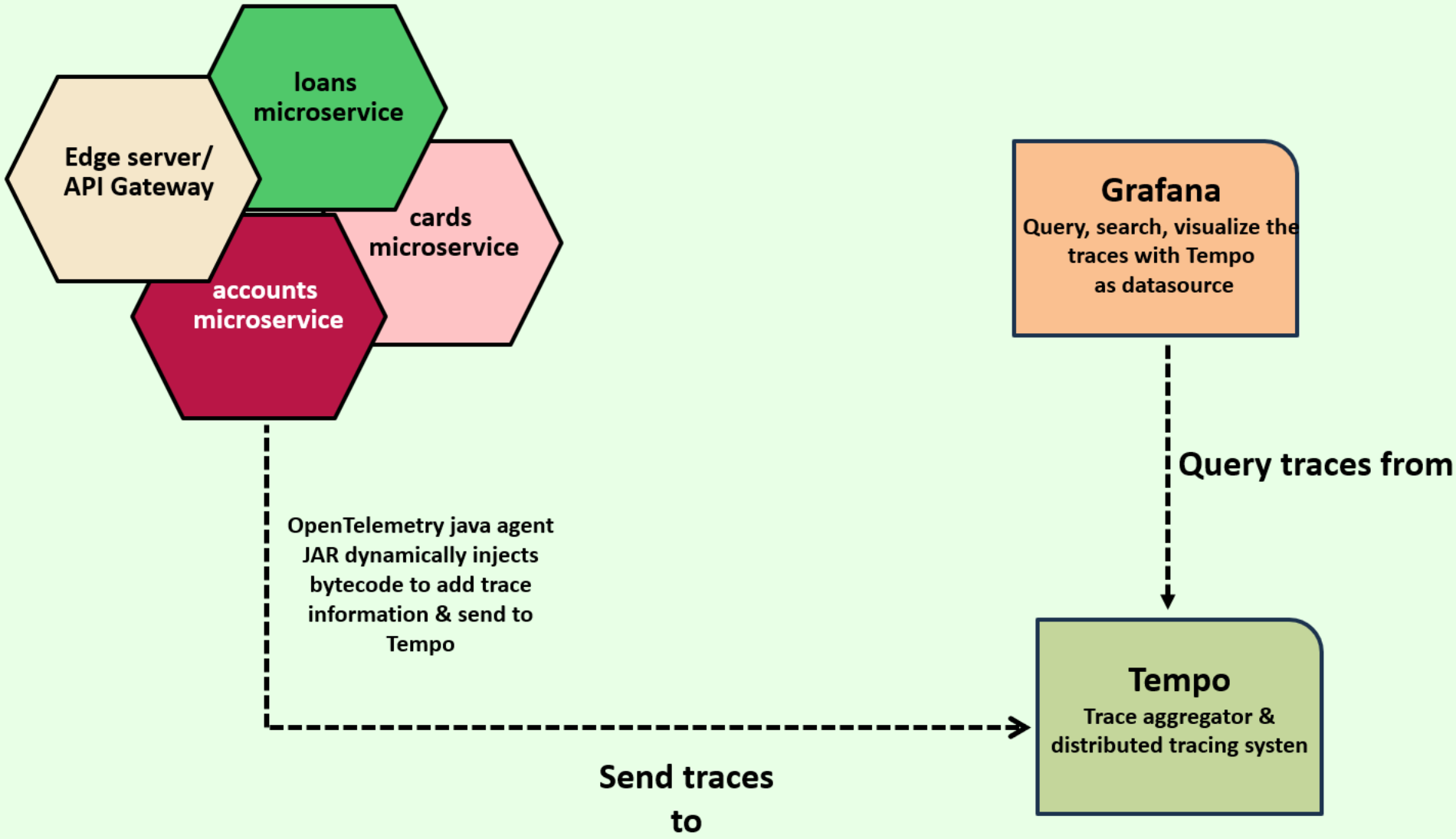


3 Grafana

Using Grafana, we can connect to Tempo as a datasource and see the distributed tracing in action with the help of visuals. We can integrate Loki and Tempo as well, so that we can jump to tracing details directly from logs inside Loki

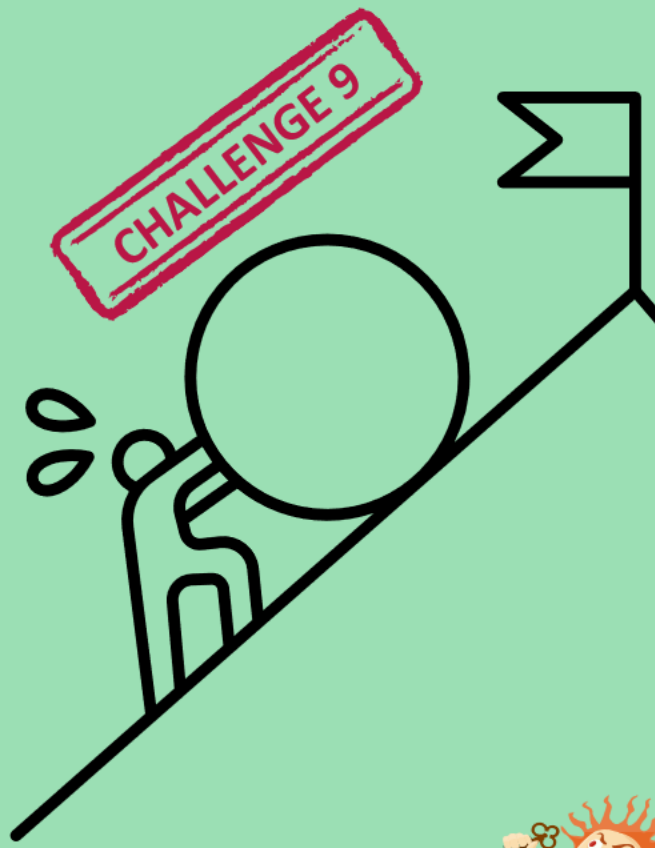
Distributed tracing with OpenTelemetry, Tempo & Grafana

eazy bytes



MICROSERVICES SECURITY

eazy bytes



SECURING MICROSERVICES FROM UNAUTHORIZED ACESS ?

How to secure our microservices and protect them from unauthorized access by client applications or end users ? Right now all our services doesn't have security and any one can invoke them to get a reponse which have sensitive data

AUTHENTICATION AND AUTHORIZATION

How can our microservices can authenticate and authorize users and services to access them. Our microservices should be capable of performing identification, authentication & authorization.

CENTRALIZED IDENTITY AND ACCESS MANAGEMENT (IAM)

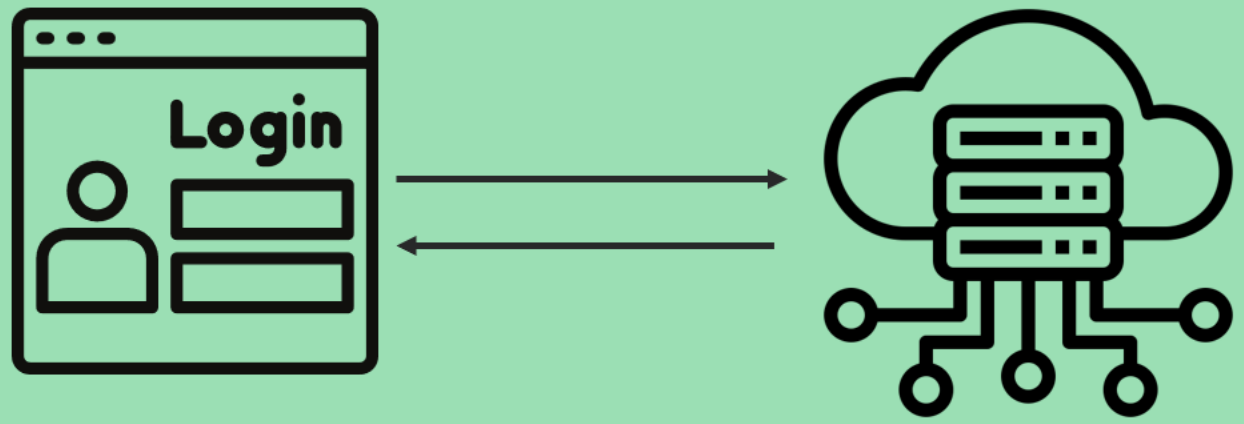
How to maintain a centralized component to store user credentials, handling identity & access management

Using OAuth2/OpenID Connect, KeyCloak (IAM), Spring Security we can secure the microservices and handle all the above challenges.

PROBLEM THAT OAUTH2 SOLVES

eazy bytes

Why should we use OAUTH2 framework for implementing security inside our microservices ? Why can't we use the basic authentication ? To answer this, first lets try to understand the basic authentication & it's drawbacks.



Early websites usually ask for credentials via an HTML form, which the browser will send to the server. The server authenticates the information and writes a session value in the cookie; as long as the session is still marked active, user can access protected features and resources.

Drawbacks of Basic authentication

Backend server or business logic is tightly coupled with the Authentication/Authorization logic. Not mobile flow/ REST APIs friendly

Basic authentication flow does not accommodate well the use case where users of one product or service would like to grant third-party clients access to their information on the platform.

PROBLEM THAT OAUTH2 SOLVES

eazy bytes

How come, Google let me use the same account in all it's products ? Though they are different websites/ Apps ?

Well the answer is with the help of OAuth2. OAuth2 recommend to use a separate Auth server for Authentication & Authorization

INTRODUCTION TO OAUTH2

eazy bytes

OAuth stands for Open Authorization. It's a free and open protocol, built on IETF standards and licenses from the Open Web Foundation. OAuth 2.1 is a security standard where you give one application permission to access your data in another application. The steps to grant permission, or consent, are often referred to as authorization or even delegated authorization. You authorize one application to access your data, or use features in another application on your behalf, without giving them your password.

Below are the few advantages of OAuth2 standard,

Supports all kinds of Apps: OAuth2 supports multiple use cases addressing different device capabilities. It supports server-to-server apps, browser-based apps, mobile/native apps, IoT devices and consoles/TVs. It has various authorization grant flows like Authorization Code grant, Client Credentials Grant Type etc. to support all kinds of apps communication.

Separation of Auth logic: Inside OAuth2, we have Authorization Server which receives requests from the Client for Access Tokens and issues them upon successful authentication. This enable us to maintain all the security logic in a single place. Regardless of how many applications an organization has, they all can connect to Auth server to perform login operation.

All user credentials & client application credentials will be maintained in a single location which is inside Auth Server.

No need to share Credentials: If you plan to allow a third-party applications and services to access your resources, then there is no need to share your credentials.

In many ways, you can think of the OAuth2 token as a "access card" at any office/hotel. These tokens provides limited access to someone, without handing over full control in the form of the master key.

OAUTH2 TERMINOLOGY

eazy bytes

Resource owner – It is you the end user. In the scenario of Stackoverflow, the end user who want to use the GitHub services to get his details. In other words, the end user owns the resources (email, profile), that's why we call him as Resource owner

Client – The website, mobile app or API will be the client as it is the one which interacts with GitHub services on behalf of the resource owner/end user. In the scenario of Stackoverflow, the Stackoverflow website is Client

Authorization Server – This is the server which knows about resource owner. In other words, resource owner should have an account in this server. In the scenario of Stackoverflow, the GitHub server which has authorization logic acts as Authorization server.

Resource Server – This is the server where the resources that client want to consume are hosted. In the scenario of Stackoverflow, the resources like User Email, Profile details are hosted inside GitHub server. So it will act as a resource server.

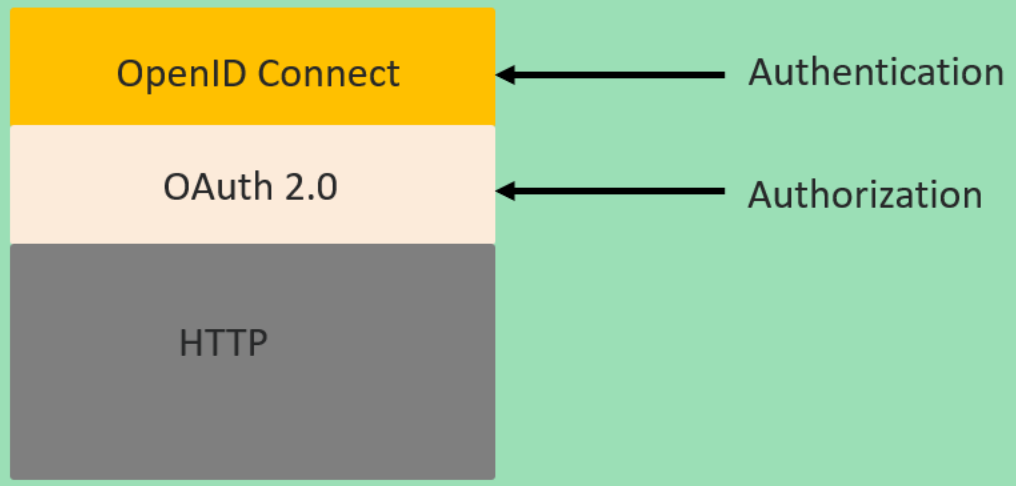
Scopes – These are the granular permissions the Client wants, such as access to data or to perform certain actions. The Auth server can issue an access token to client with the scope of Email, READ etc.

WHAT IS OPENID CONNECT & WHY IT IS IMPORTANT ?

eazy bytes

What is OpenID Connect?

- OpenID Connect is a protocol that sits on top of the OAuth 2.0 framework. While OAuth 2.0 provides authorization via an access token containing scopes, OpenID Connect provides authentication by introducing a new ID token which contains a new set of information and claims specifically for identity.
- With the ID token, OpenID Connect brings standards around sharing identity details among the applications.



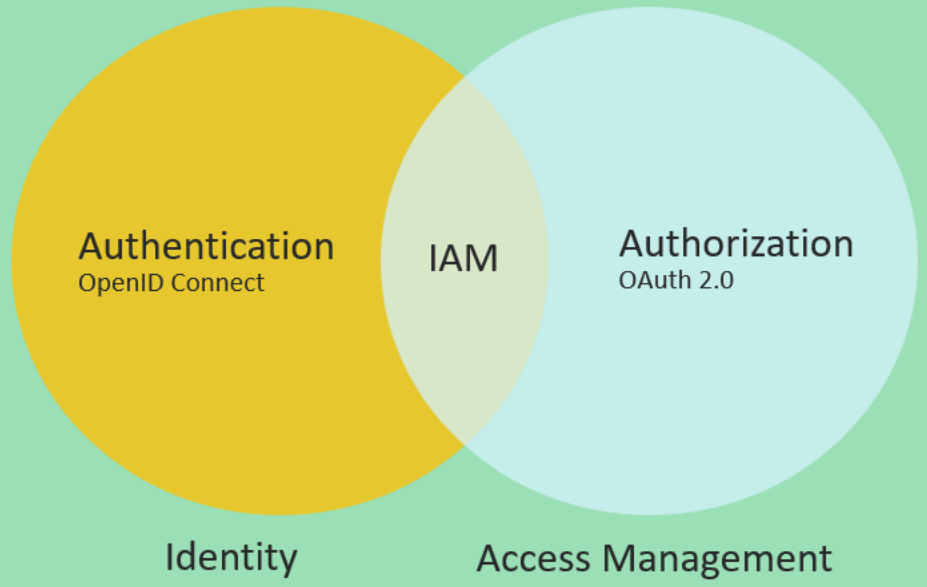
The OpenID Connect flow looks the same as OAuth. The only differences are, in the initial request, a specific scope of openid is used, and in the final exchange the client receives both an Access Token and an ID Token.

WHAT IS OPENID CONNECT & WHY IT IS IMPORTANT ?

eazy bytes

Why is OpenID Connect important?

- Identity is the key to any application. At the core of modern authorization is OAuth 2.0, but OAuth 2.0 lacks an authentication component. Implementing OpenID Connect on top of OAuth 2.0 completes an IAM (Identity & Access Management) strategy.
- As more and more applications need to connect with each other and more identities are being populated on the internet, the demand to be able to share these identities is also increased. With OpenID connect, applications can share the identities easily and standard way.

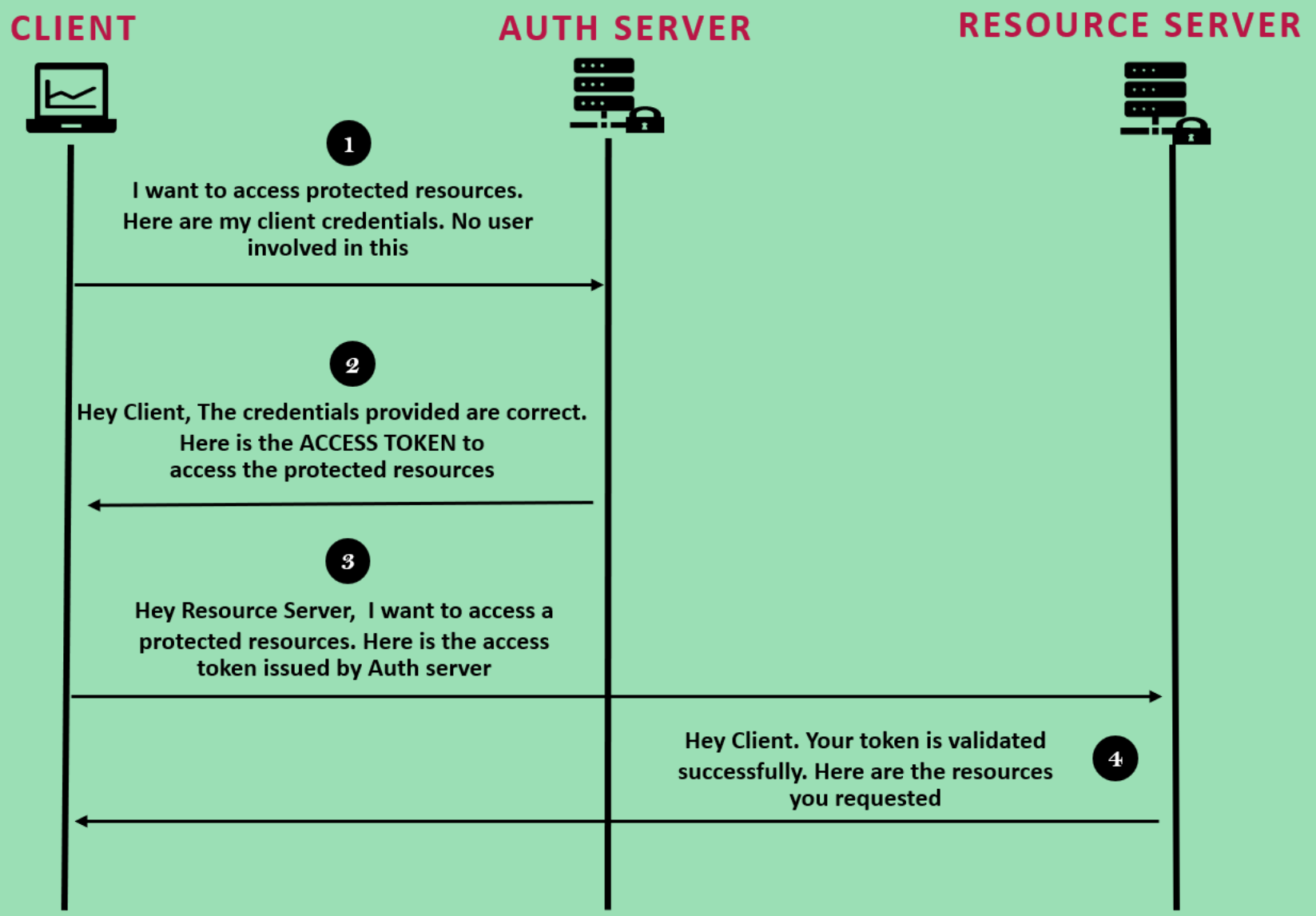


OpenID Connect add below details to OAuth 2.0

1. OIDC standardizes the scopes to openid, profile, email, and address.
2. ID Token using JWT standard
3. OIDC exposes the standardized “/userinfo” endpoint.

CLIENT CREDENTIALS GRANT TYPE FLOW IN OAUTH2

eazy bytes



CLIENT CREDENTIALS GRANT TYPE FLOW IN OAUTH2

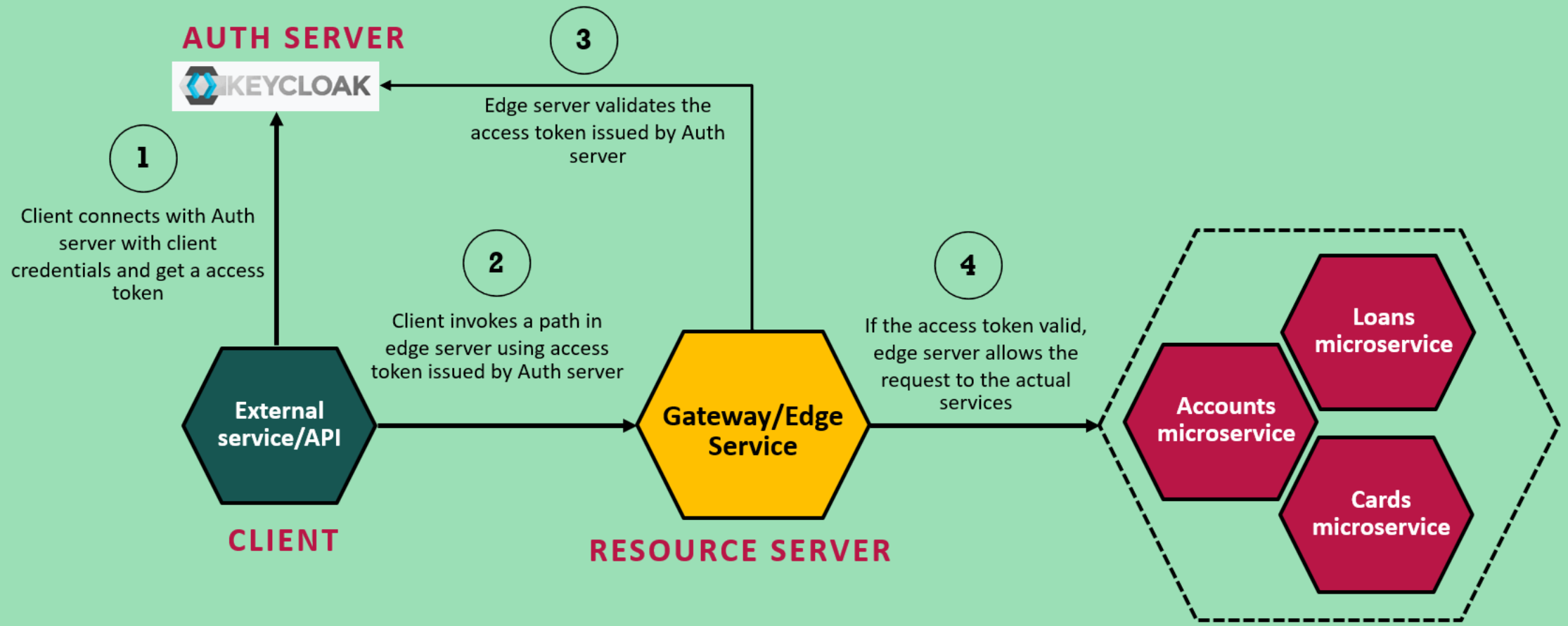
eazy bytes

- ✓ In the step 1, where client is making a request to Auth Server endpoint, have to send the below important details,
  - client_id & client_secret – the credentials of the client to authenticate itself.
  - scope – similar to authorities. Specifies level of access that client is requesting like EMAIL, PROFILE
  - grant_type – With the value 'client_credentials' which indicates that we want to follow client credentials grant type

- ✓ This is the most simplest grant type flow in OAUTH2.
- ✓ We use this authentication flow only if there is no user and UI involved. Like in the scenarios where 2 different applications want to share data between them using backend APIs.

SECURING GATEWAY USING CLIENT CREDENTIALS GRANT TYPE FLOW IN OAUTH2

eazy bytes

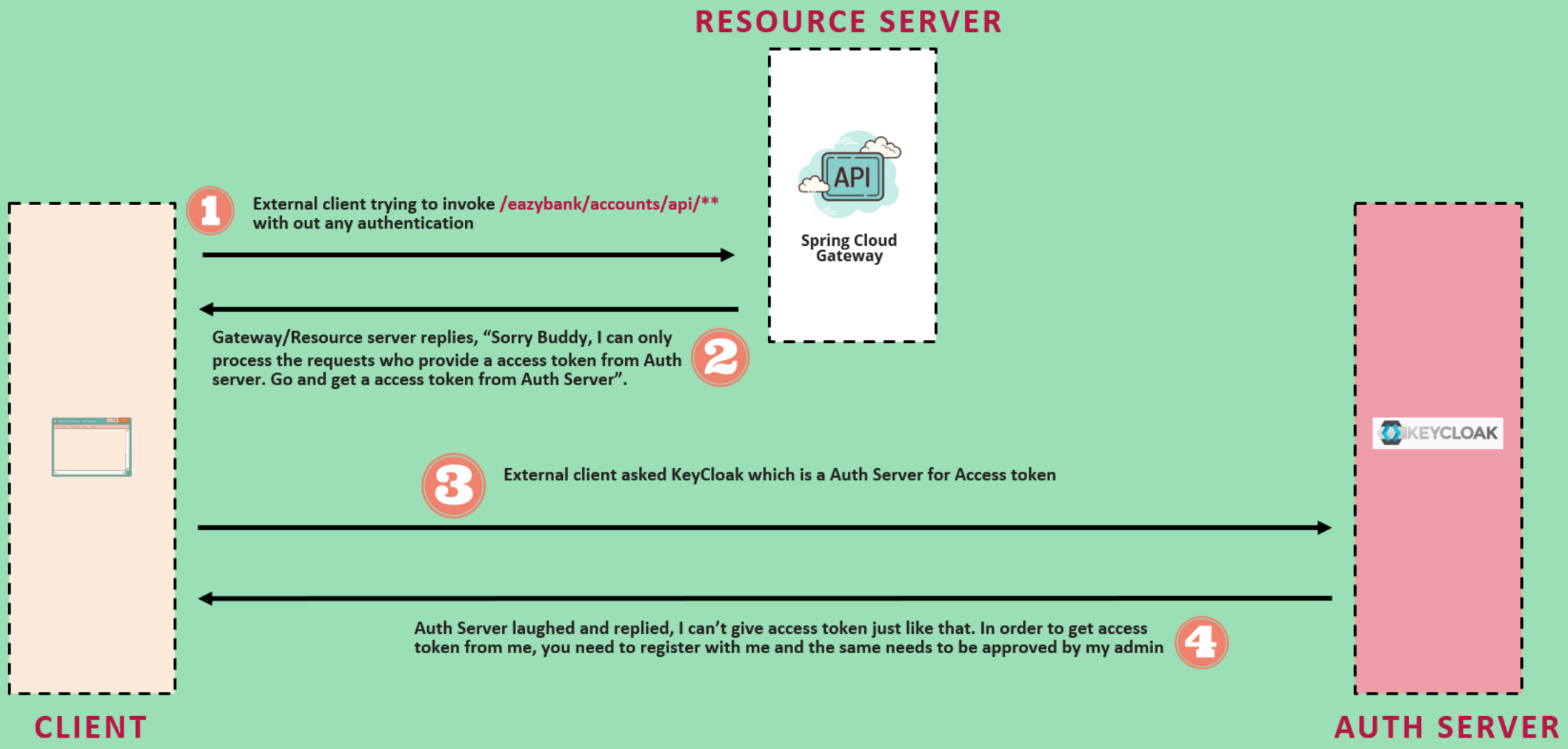


Unsecured services deployed behind the docker network or Kubernetes firewall network. So can't be accessed directly

*** When ever an external system trying to communicate with Spring Cloud Gateway where there is no end user involved, then we need to use the OAuth2 Client Credentials grant flow for Authentication & Authorization.**

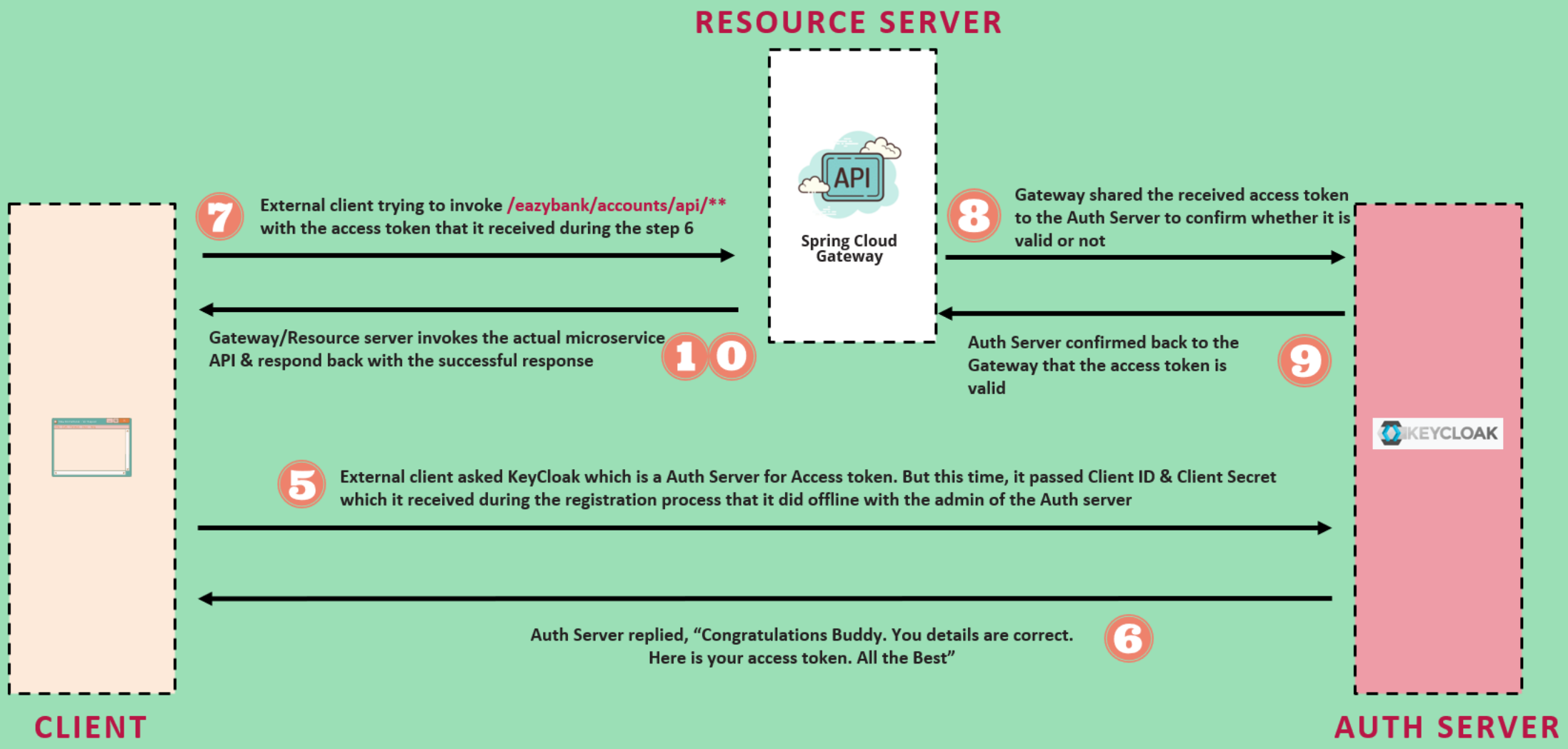
SECURING GATEWAY USING CLIENT CREDENTIALS GRANT TYPE FLOW IN OAUTH2

eazy bytes



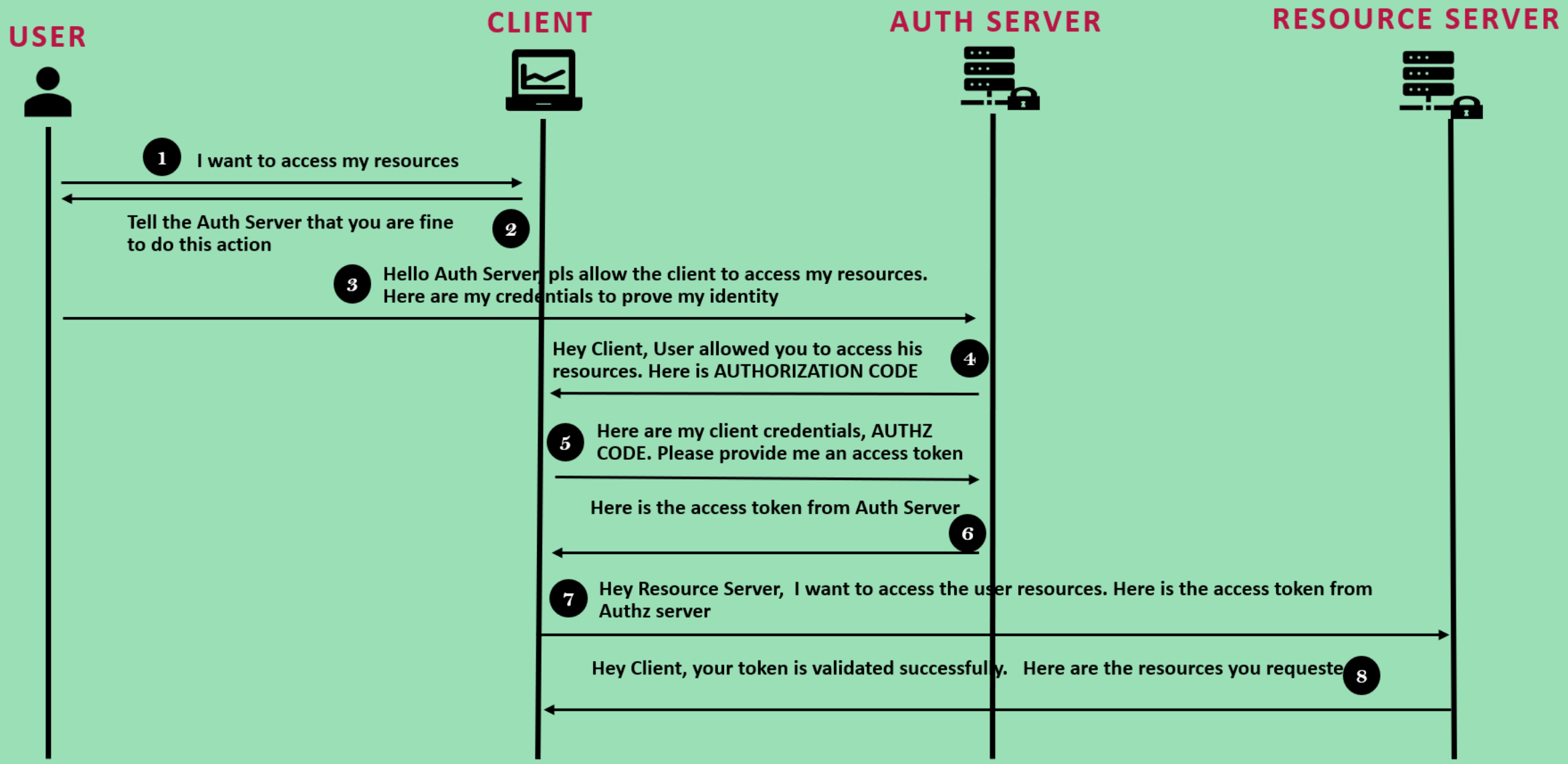
SECURING GATEWAY USING CLIENT CREDENTIALS GRANT TYPE FLOW IN OAUTH2

eazy bytes



AUTHORIZATION CODE GRANT TYPE FLOW IN OAUTH2

eazy bytes



AUTHORIZATION CODE GRANT TYPE FLOW IN OAUTH2

eazy bytes

✓ In the steps 2 & 3, where client is making a request to Auth Server endpoint have to send the below important details,

- **client_id** – the id which identifies the client application by the Auth Server. This will be granted when the client register first time with the Auth server.
- **redirect_uri** – the URI value which the Auth server needs to redirect post successful authentication. If a default value is provided during the registration then this value is optional
- **scope** – similar to authorities. Specifies level of access that client is requesting like READ
- **state** – CSRF token value to protect from CSRF attacks
- **response_type** – With the value 'code' which indicates that we want to follow authorization code grant

✓ In the step 5 where client after received a authorization code from Auth server, it will again make a request to Auth server for a token with the below values,

- **code** – the authorization code received from the above steps
- **client_id & client_secret** – the client credentials which are registered with the auth server. Please note that these are not user credentials
- **grant_type** – With the value 'authorization_code' which identifies the kind of grant type is used
- **redirect_uri**

AUTHORIZATION CODE GRANT TYPE FLOW IN OAUTH2

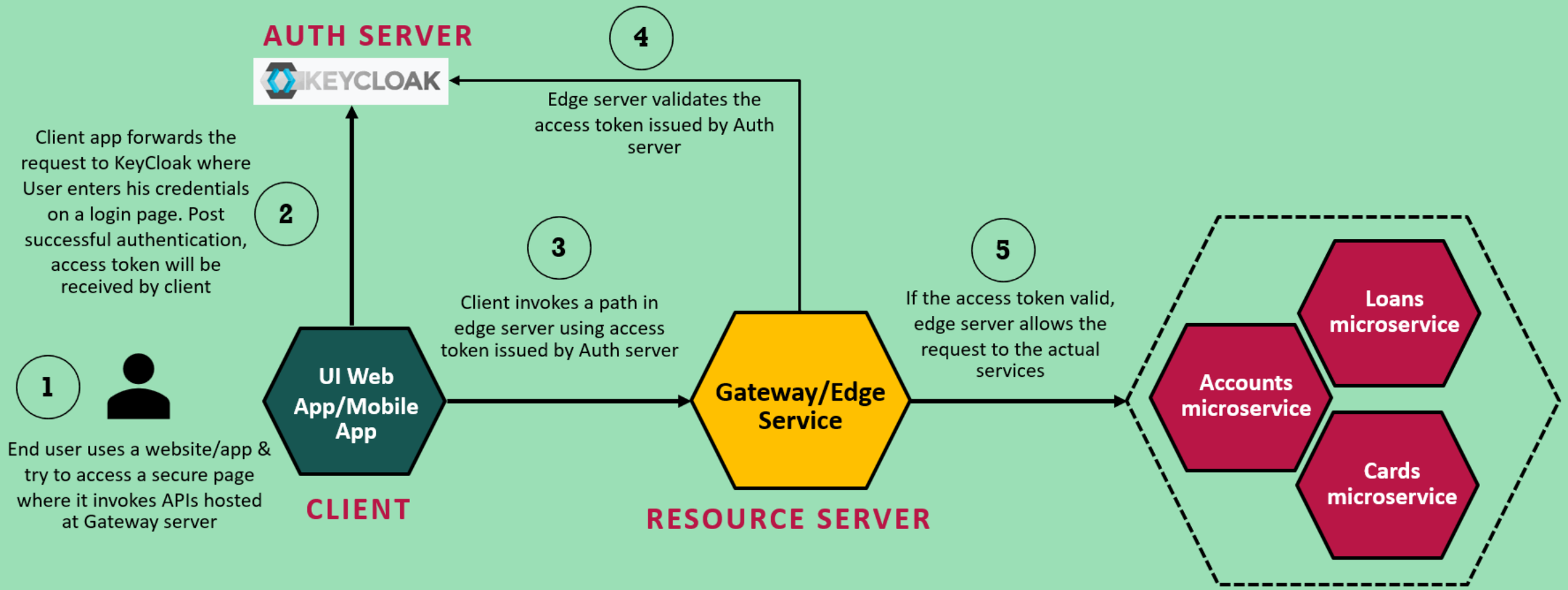
eazy bytes

- ✓ We may wonder that why in the Authorization Code grant type client is making request 2 times to Auth server for authorization code and access token.
  - In the first step, authorization server will make sure that user directly interacted with it along with the credentials. If the details are correct, auth server send the authorization code to client
  - Once it receives the authorization code, in this step client has to prove it's identity along with the authorization code & client credentials to get the access token.

- ✓ Well you may ask why can't Auth server directly club both the steps together and provide the token in a single step. The answer is that we used to have that grant type as well which is called as **'implicit grant type'**. But this grant type is deprecated as it is less secure.

SECURING GATEWAY USING AUTHORIZATION CODE GRANT TYPE FLOW IN OAUTH2

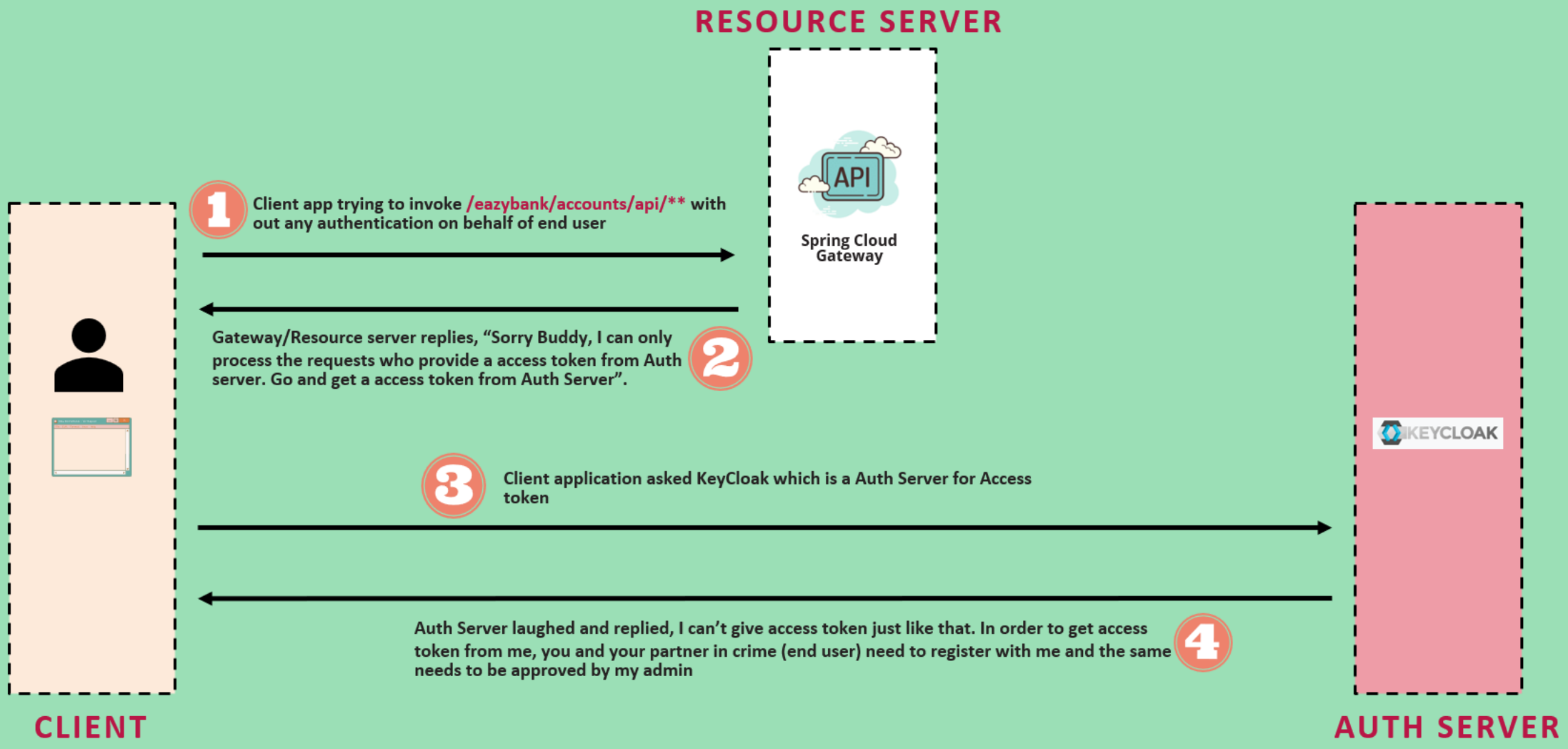
eazy bytes



** When ever an end user involved & trying to communicate with Spring Cloud Gateway, then we need to use the OAuth2 Authorization Code grant flow for Authentication & Authorization.

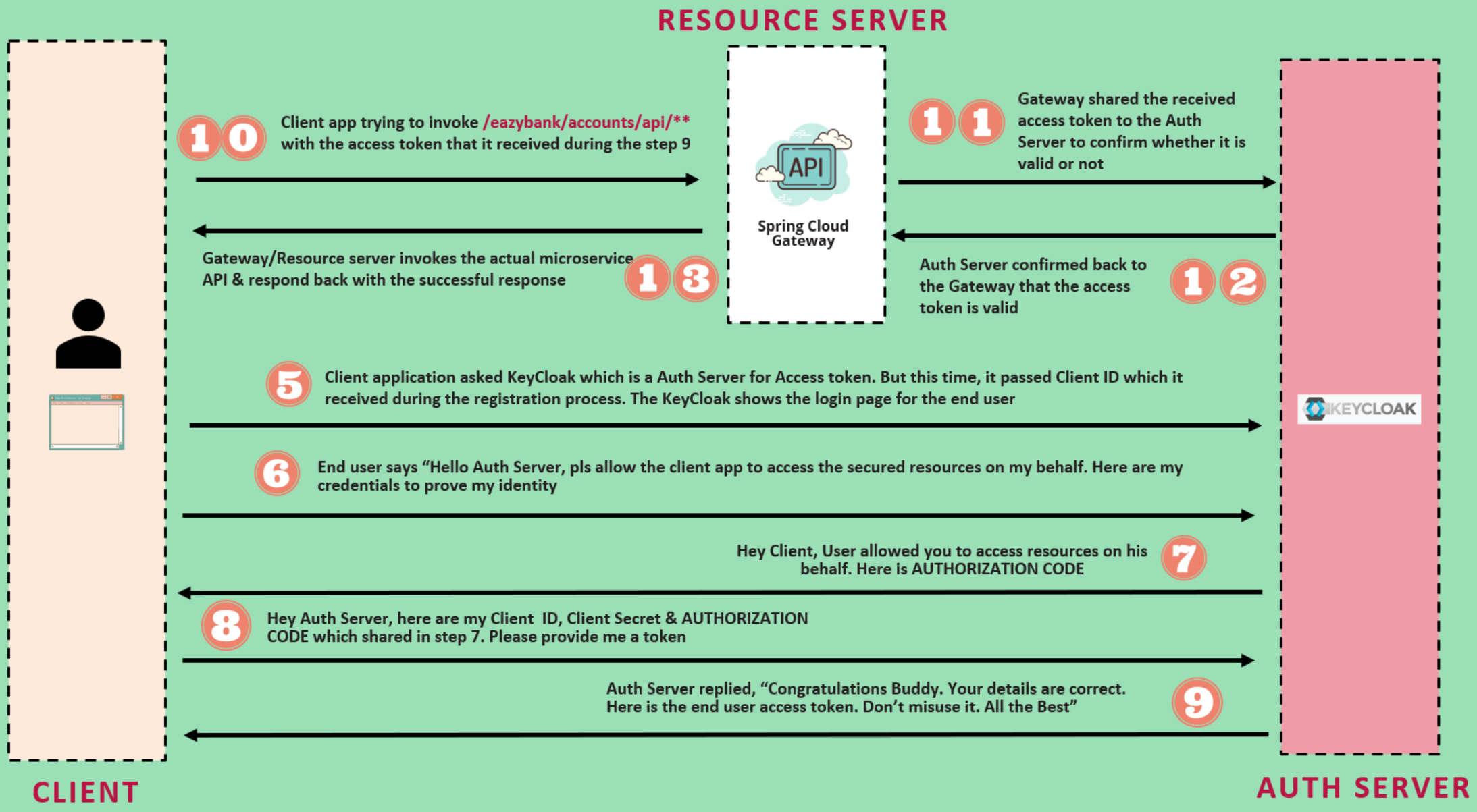
SECURING GATEWAY USING AUTHORIZATION CODE GRANT TYPE FLOW IN OAUTH2

eazy bytes



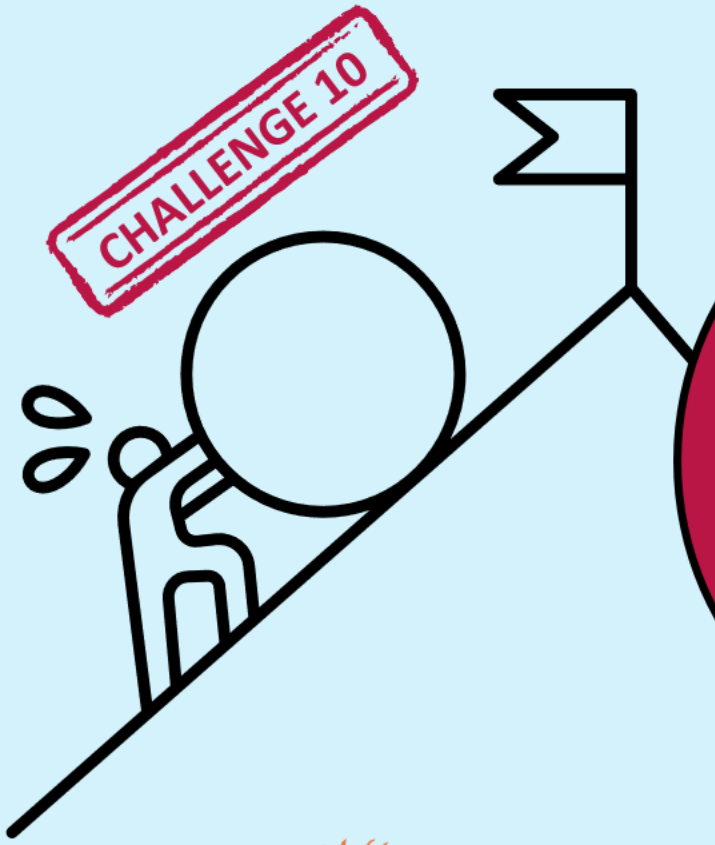
SECURING GATEWAY USING AUTHORIZATION CODE GRANT TYPE FLOW IN OAUTH2

eazy bytes



Building Event-driven microservices

eazy bytes



Avoiding temporal coupling whenever possible

Temporal coupling occurs when a caller service expects an immediate response from a callee service before continuing its processing. If the callee experiences any delay in responding, it negatively impacts the overall response time of the caller. This scenario commonly arises in synchronous communication between services. How can we prevent temporal coupling and mitigate its effects?

Using asynchronous communication

Synchronous communication between services is not always necessary. In many real-world scenarios, asynchronous communication can fulfill the requirements effectively. So, how can we establish asynchronous communication between services?

Building event driven microservices

An event, as an incident, signifies a significant occurrence within a system, such as a state transition. Multiple sources can generate events. When an event takes place, it is possible to alert the concerned parties. How can one go about constructing event-driven services with these characteristics?

Event-driven microservices can be built using Event-driven architecture, producing and consuming events using Async communication, event brokers, Spring Cloud Function, Spring Cloud Stream. Let's explore the world of event-driven microservices

Event-driven models

eazy bytes

Event-driven architectures can be built using two primary models

Publisher/Subscriber (Pub/Sub) Model

This model revolves around subscriptions. Producers generate events that are distributed to all subscribers for consumption. Once an event is received, it cannot be replayed, which means new subscribers joining later will not have access to past events.

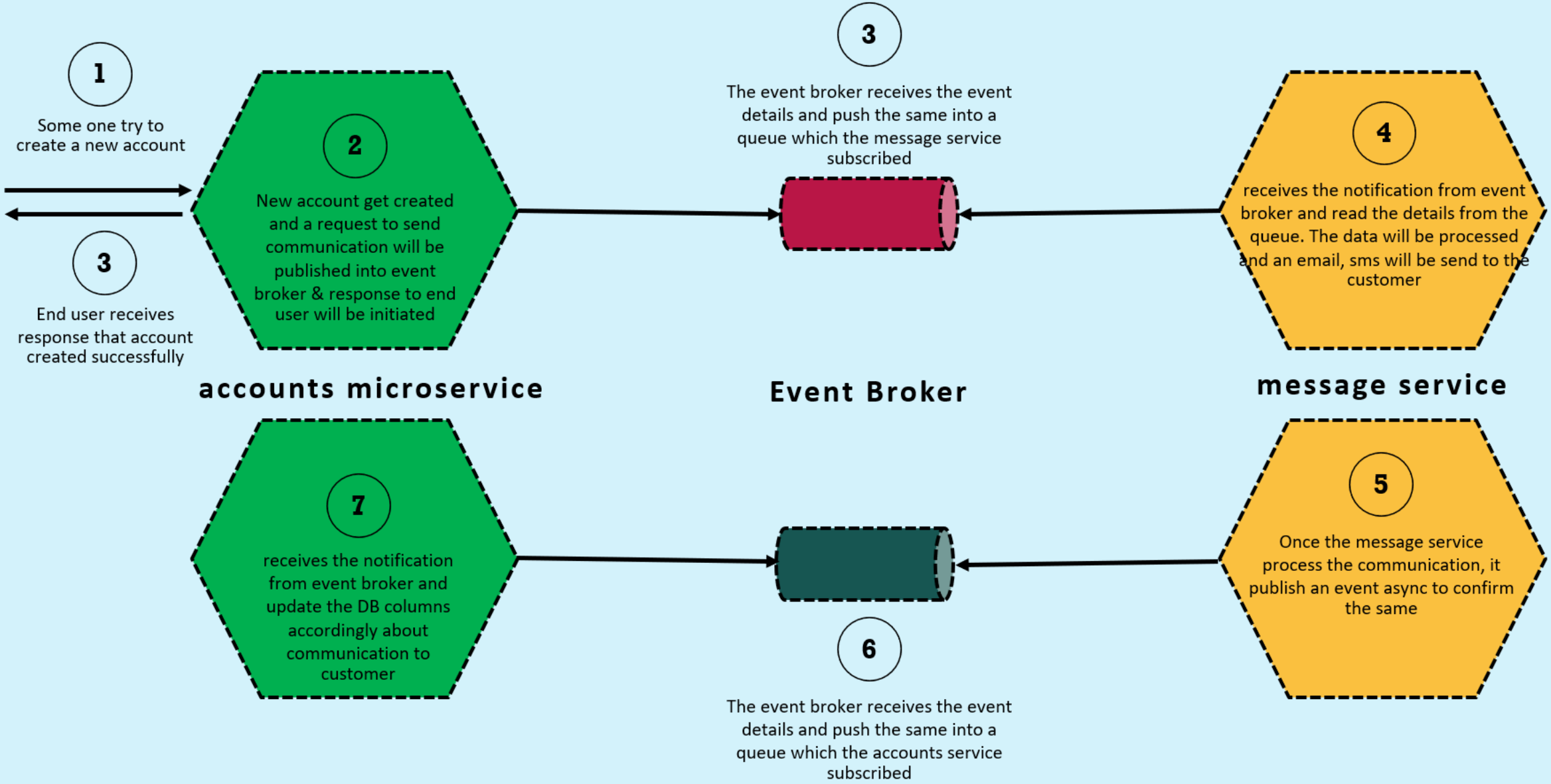
Event Streaming Model

In this model, events are written to a log in a sequential manner. Producers publish events as they occur, and these events are stored in a well-ordered fashion. Instead of subscribing to events, consumers have the ability to read from any part of the event stream. One advantage of this model is that events can be replayed, allowing clients to join at any time and receive all past events.

The pub/sub model is frequently paired with **RabbitMQ** as a popular option. On the other hand, Apache **Kafka** is a robust platform widely utilized for event stream processing.

What we are going to build using a pub/sub model

eazy bytes



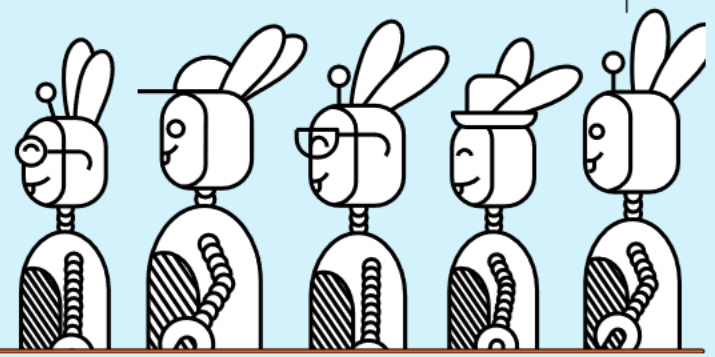
Using RabbitMQ for publish/subscribe communications

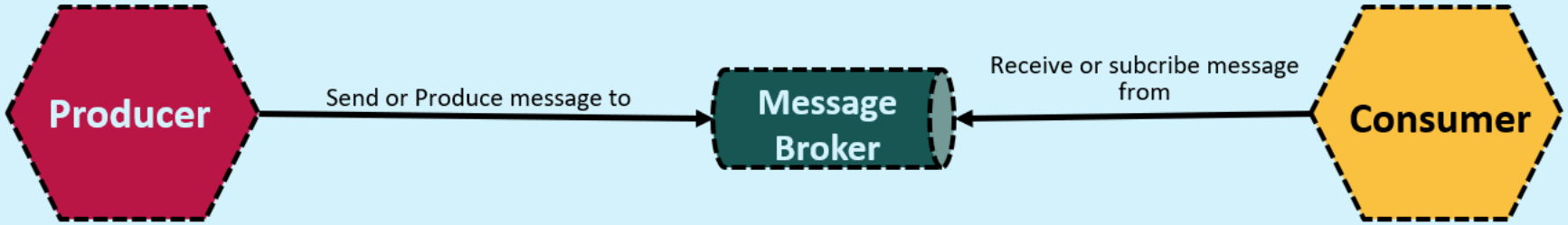
eazy bytes

RabbitMQ, an open-source message broker, is widely recognized for its utilization of AMQP (Advanced Message Queuing Protocol) and its ability to offer flexible asynchronous messaging, distributed deployment, and comprehensive monitoring. Furthermore, recent versions of RabbitMQ have incorporated event streaming functionalities into their feature set.

When using an AMQP-based solution such as RabbitMQ, the participants engaged in the interaction can be classified into the following categories:

- **Producer:** The entity responsible for sending messages (also known as the publisher).
- **Consumer:** The entity tasked with receiving messages (also known as the subscriber).
- **Message broker:** The middleware that receives messages from producers and directs them to the appropriate consumers.

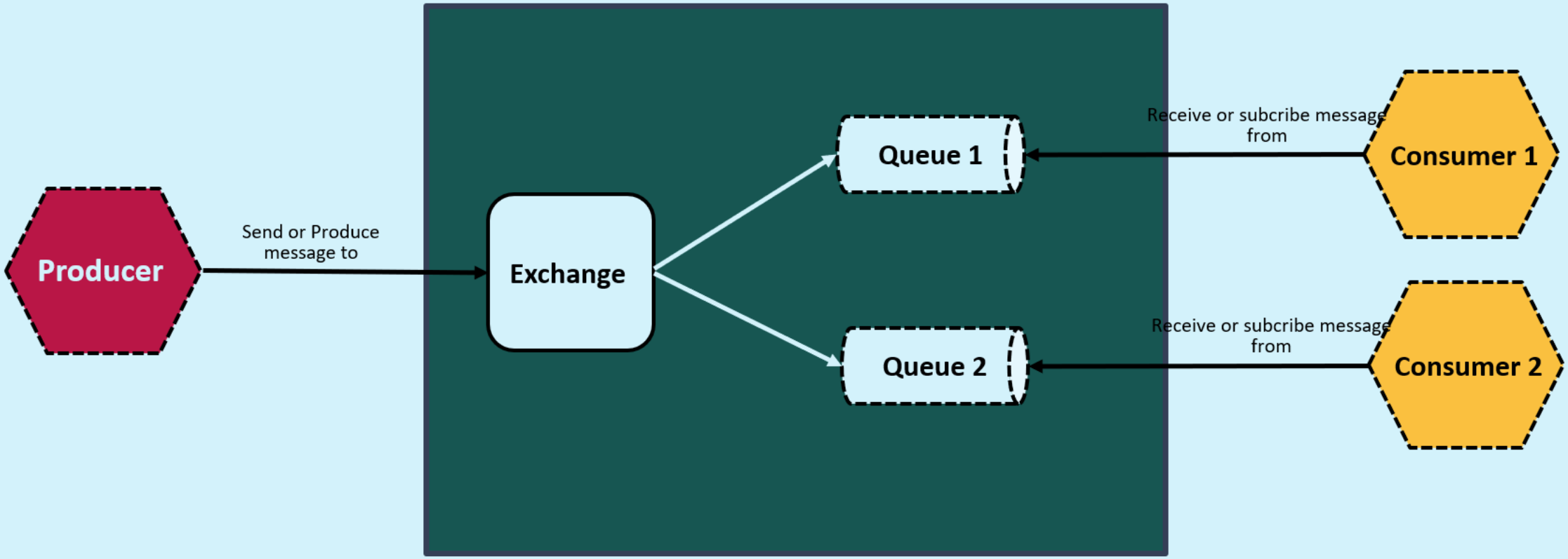




Using RabbitMQ for publish/subscribe communications

eazy bytes

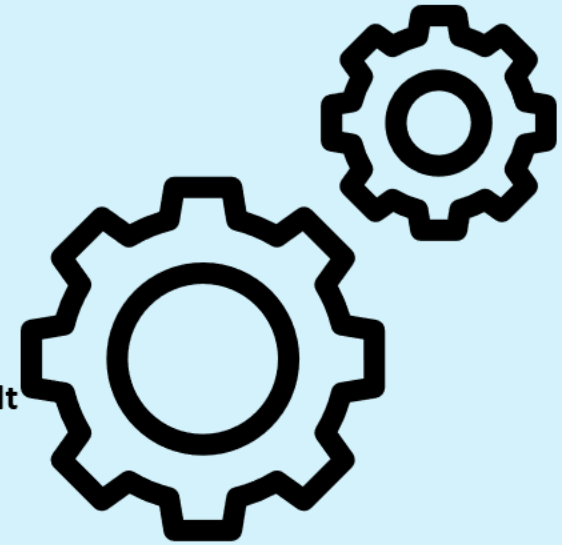
The messaging model of AMQP operates on the principles of **exchanges** and **queues**, as depicted in the following illustration. Producers transmit messages to an exchange. Based on a specified routing rule, RabbitMQ determines the queues that should receive a copy of the message. Consumers, in turn, read messages from a queue.



Why to use Spring Cloud Function ?

eazy bytes

Spring Cloud Function facilitates the development of business logic by utilizing functions that adhere to the standard interfaces introduced in Java 8, namely **Supplier**, **Function**, and **Consumer**.



- **Supplier:** A supplier is a function that produces an output without requiring any input. It can also be referred to as a producer, publisher, or source.
- **Function:** A function accepts input and generates an output. It is commonly referred to as a processor.
- **Consumer:** A consumer is a function that consumes input but does not produce any output. It can also be called a subscriber or sink.

Spring Cloud Function features:

- Choice of programming styles - reactive, imperative or hybrid.
- POJO functions (i.e., if something fits the @FunctionalInterface semantics we'll treat it as function)
- Function composition which includes composing imperative functions with reactive.
- REST support to expose functions as HTTP endpoints etc.
- Streaming data (via Apache Kafka, Solace, RabbitMQ and more) to/from functions via Spring Cloud Stream framework.
- Packaging functions for deployments, specific to the target platform (e.g., AWS Lambda and possibly other "serverless" service providers)

Steps to create functions using Spring Cloud Functions

eazy bytes

Below are the steps to create functions using Spring Cloud Functions,

1 **Initialize a spring cloud function project:** Start by creating a new Spring Boot project using your preferred IDE or by using Spring Initializr (https://start.spring.io/). Include the **spring-cloud-function-context** maven dependency

2 **Implement the business logic using functions**

Develop two functions with the name email() and sms() like in the image. To make it simple, for now they just have logic of logging the details. But in real projects you can write logic to send emails and messages.

To enable Spring Cloud Function to recognize our functions, we need to register them as beans. Proceed with annotating the MessageFunctions class as @Configuration and the methods email() & sms() as @Bean to accomplish this.

```java
@Configuration
public class MessageFunctions {

    private static final Logger log = LoggerFactory.getLogger(MessageFunctions.class);

    @Bean
    public Function<AccountsMsgDto, AccountsMsgDto> email() {
        return accountsMsgDto -> {
            log.info("Sending email with the details : " +  accountsMsgDto.toString());
            return accountsMsgDto;
        };
    }

    @Bean
    public Function<AccountsMsgDto, Long> sms() {
        return accountsMsgDto -> {
            log.info("Sending sms with the details : " + accountsMsgDto.toString());
            return accountsMsgDto.accountNumber();
        };
    }
```

Steps to create functions using Spring Cloud Functions

eazy bytes

3 **Composing functions:** If our scenario needs multiple functions to be executed, then we need to compose them otherwise we can use them as individual functions as well. Composing functions can be achieved by defining a property in application.yml like shown below,

```
spring:
  cloud:
    function:
      definition: email|sms
```

The property spring.cloud.function.definition enables you to specify which functions should be managed and integrated by Spring Cloud Function, thereby establishing a specific data flow. In the previous step, we implemented the email() and sms() functions. We can now instruct Spring Cloud Function to utilize these functions as building blocks and generate a new function derived from their composition.

In serverless applications designed for deployment on FaaS platforms like AWS Lambda, Azure Functions, Google Cloud Functions, or Knative, it is common to have one function defined per application. The definition of cloud functions can align directly with functions declared in your application on a one-to-one basis. Alternatively, you can employ the pipe (|) operator to compose functions together in a data flow. In cases where you need to define multiple functions, the semicolon (;) character can be used as a separator instead of the pipe (|).

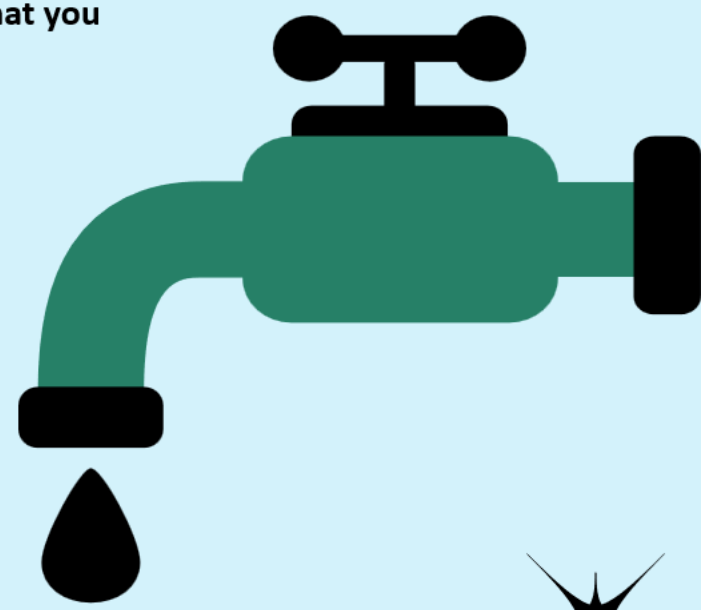
Based on the provided functions, the framework offers various ways to expose them according to our needs. For instance, Spring Cloud Function can automatically expose the functions specified in spring.cloud.function.definition as REST endpoints. This allows you to package the application, deploy it on a FaaS platform such as Knative, and instantly have a serverless Spring Boot application. But that is not what we want. Moving forward, the next step involves integrating it with **Spring Cloud Stream** and binding the function to message channels within an event broker like RabbitMQ.

Why to use Spring Cloud Stream ?

eazy bytes

Spring Cloud Stream is a framework designed for creating scalable, event-driven, and streaming applications. Its core principle is to allow developers to focus on the business logic while the framework takes care of infrastructure-related tasks, such as integrating with a message broker.

Spring Cloud Stream leverages the native capabilities of each message broker, while also providing an abstraction layer to ensure a consistent experience regardless of the underlying middleware. By just adding a dependency to your project, you can have functions automatically connected to an external message broker. The beauty of this approach is that you don't need to modify any application code; you simply adjust the configuration in the application.yml file.



The framework supports integrations with RabbitMQ, Apache Kafka, Kafka Streams, and Amazon Kinesis. There are also integrations maintained by partners for Google PubSub, Solace PubSub+, Azure Event Hubs, and Apache RocketMQ.

The core building blocks of Spring Cloud Stream are:

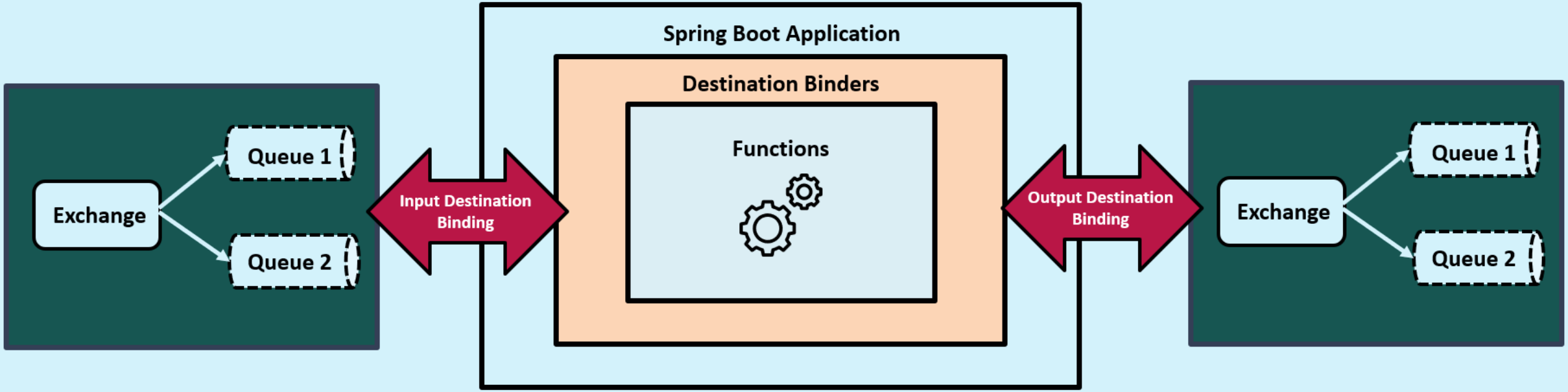
- **Destination Binders:** Components responsible to provide integration with the external messaging systems.
- **Destination Bindings:** Bridge between the external messaging systems and application code (producer/consumer) provided by the end user.
- **Message:** The canonical data structure used by producers and consumers to communicate with Destination Binders (and thus other applications via external messaging systems).

Why to use Spring Cloud Stream ?

eazy bytes

Spring Cloud Stream equips a Spring Boot application with a destination binder that seamlessly integrates with an external messaging system. This binder takes on the responsibility of establishing communication channels between the application's producers and consumers and the entities within the messaging system (such as exchanges and queues in the case of RabbitMQ). These communication channels, known as destination bindings, serve as connections between applications and brokers.

A destination binding can function as either an input channel or an output channel. By default, Spring Cloud Stream maps each binding, both input and output, to an exchange within RabbitMQ (specifically, a topic exchange). Additionally, for each input binding, it binds a queue to the associated exchange. This queue serves as the source from which consumers receive and process events. This configuration provides the necessary infrastructure for implementing event-driven architectures based on the pub/sub model.



Steps to create bindings using Spring Cloud Stream

eazy bytes

Below are the steps to create bindings using Spring Cloud Stream,

1 **Add the Stream related dependencies:** Add the maven dependencies spring-cloud-stream, spring-cloud-stream-binder-rabbit inside pom.xml of message service where we defined functions

2 **Add the stream binding and rabbitmq properties inside application.yml of message service**

We need to define input binding for each function accepting input data, and an output binding for each function returning output data. Each binding can have a logical name following the below convention. Unless you use partitions (for example, with Kafka), the <index> part of the name will always be 0. The <functionName> is computed from the value of the spring.cloud.function.definition property.

Input binding: <functionName> + -in- + <index>
Output binding: <functionName> + -out- + <index>

The binding names exist only in Spring Cloud Stream and RabbitMQ doesn't know about them. So to map between the Spring Cloud Stream binding and RabbitMQ, we need to define destination which will be the exchange inside the RabbitMQ. group is typically application name, so that all the instances of the application can point to same exchange and queue.

The queues will be created inside RabbitMQ based on the queue-naming strategy (<destination>.<group>) includes a parameter called consumer group.

```
spring:
  application:
    name: message
  cloud:
    function:
      definition: email|sms
    stream:
      bindings:
        emailsms-in-0:
          destination: send-communication
          group: ${spring.application.name}
        emailsms-out-0:
          destination: communication-sent
  rabbitmq:
    host: localhost
    port: 5672
    username: guest
    password: guest
    connection-timeout: 10s
```

Event producing and consuming in accounts microservice

eazy bytes

Below are the steps for event producing and consuming in accounts microservice

1 **Autowire StreamBridge class:** StreamBridge is a class inside Spring Cloud Stream which allows user to send data to an output binding. So to produce the event, autowire the StreamBridge class into the class from where you want to produce a event

2 **Use send() of StreamBridge to produce a event like shown below,**

```
@Override
public void createAccount(CustomerDto customerDto) {
    Customer customer = CustomerMapper.mapToCustomer(customerDto, new Customer());
    Optional<Customer> optionalCustomer = customerRepository.findByMobileNumber(
            customerDto.getMobileNumber());
    if (optionalCustomer.isPresent()) {
        throw new CustomerAlreadyExistsException("Customer already registered with given mobileNumber "
                + customerDto.getMobileNumber());
    }
    Customer savedCustomer = customerRepository.save(customer);
    Accounts savedAccount = accountsRepository.save(createNewAccount(savedCustomer));
    sendCommunication(savedAccount, savedCustomer);
}

private void sendCommunication(Accounts account, Customer customer) {
    var accountsMsgDto = new AccountsMsgDto(account.getAccountNumber(), customer.getName(),
            customer.getEmail(), customer.getMobileNumber());
    log.info("Sending Communication request for the details: {}", accountsMsgDto);
    var result = streamBridge.send("sendCommunication-out-0", accountsMsgDto);
    log.info("Is the Communication request successfully processed ? : {}", result);
}
```

Event producing and consuming in accounts microservice

eazy bytes

3

Create a function to accept the event: Inside accounts microservice, we need to create a function that accepts the event and update the communication status inside the DB. Below is a sample code snippet of the same

```
@Configuration
public class AccountsFunctions {

   private static final Logger log = LoggerFactory.getLogger(AccountsFunctions.class);

   @Bean
   public Consumer<Long> updateCommunication(IAccountsService accountsService) {
      return accountNumber -> {
         log.info("Updating Communication status for the account number : " + accountNumber.toString());
         accountsService.updateCommunicationStatus(accountNumber);
      };
   }
```

Event producing and consuming in accounts microservice

eazy bytes

4 **Add the stream binding and rabbitmq properties inside application.yml of accounts service**

when accounts microservice want to produce a event using StreamBridge, we should have a supporting stream binding and destination. The same we created with the names `sendCommunication-out-0 and send-communication`

Similarly we need to define input binding for the function `updateCommunication` to accept the event using the destination `communication-sent.` So when the message service push a event into the exchange of `communication-sent,` the same will be processed by the function updateCommunication

```yaml
spring:
  application:
    name: "accounts"
  cloud:
    function:
      definition: updateCommunication
    stream:
      bindings:
        updateCommunication-in-0:
          destination: communication-sent
          group:  ${spring.application.name}
        sendCommunication-out-0:
          destination: send-communication
  rabbitmq:
    host: localhost
    port: 5672
    username: guest
    password: guest
    connection-timeout: 10s
```

Apache Kafka Vs RabbitMQ

eazy bytes

Kafka and RabbitMQ are both popular messaging systems, but they have some fundamental differences in terms of design philosophy, architecture, and use cases. Here are the key distinctions between Kafka and RabbitMQ:

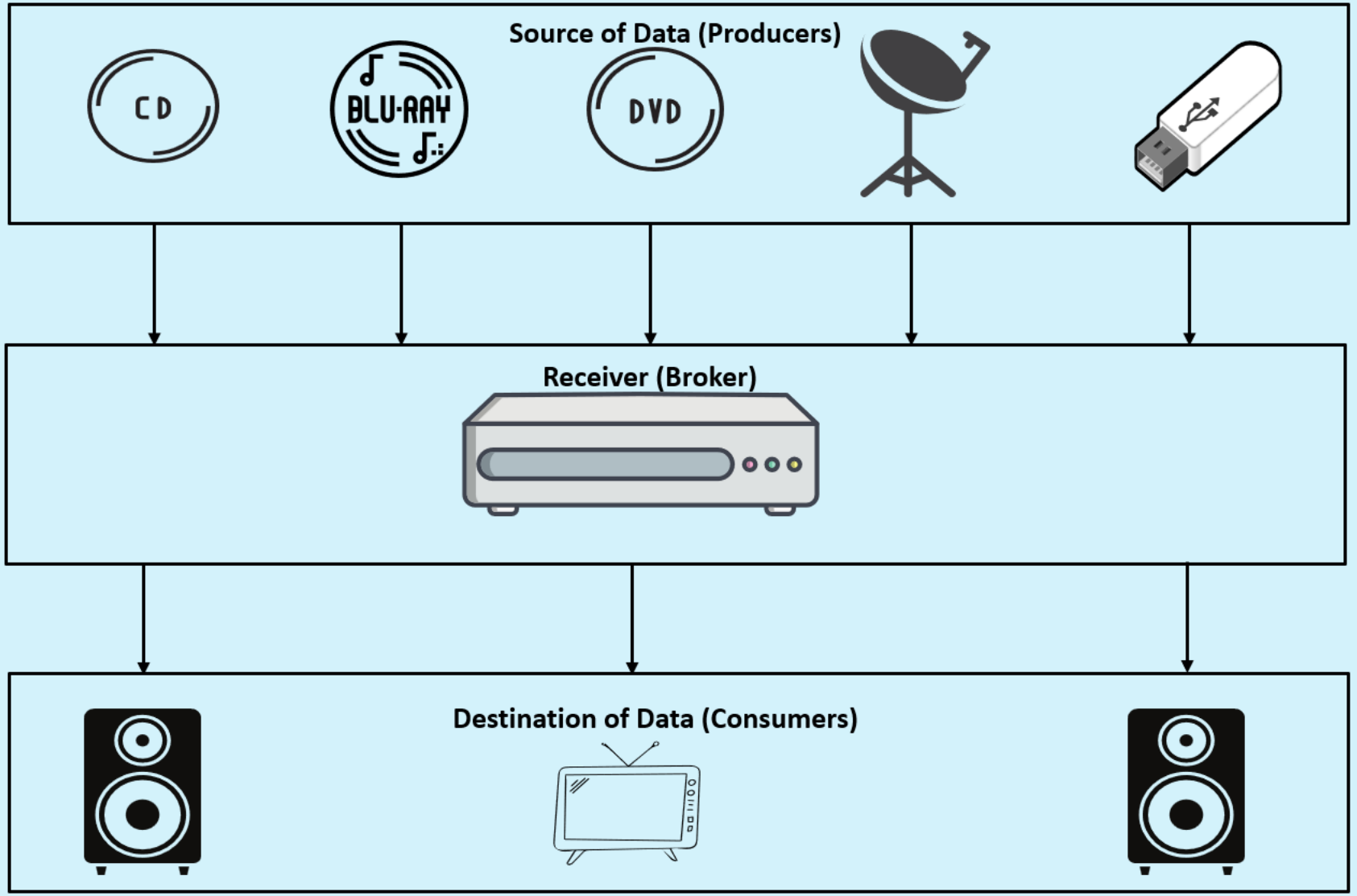
- **Design:** Kafka is a distributed event streaming platform, while RabbitMQ is a message broker. This means that Kafka is designed to handle large volumes of data, while RabbitMQ is designed to handle smaller volumes of data with more complex routing requirements.

- **Data retention:** Kafka stores data on disk, while RabbitMQ stores data in memory. This means that Kafka can retain data for longer periods of time, while RabbitMQ is more suitable for applications that require low latency.

- **Performance:** Kafka is generally faster than RabbitMQ, especially for large volumes of data. However, RabbitMQ can be more performant for applications with complex routing requirements.

- **Scalability:** Kafka is highly scalable, while RabbitMQ is more limited in its scalability. This is because Kafka can be scaled horizontally to any extent by adding more brokers to the cluster

Ultimately, the best choice for you will depend on your specific needs and requirements. If you need a high-performance messaging system that can handle large volumes of data, Kafka is a good choice. If you need a messaging system with complex routing requirements, RabbitMQ is a good choice.

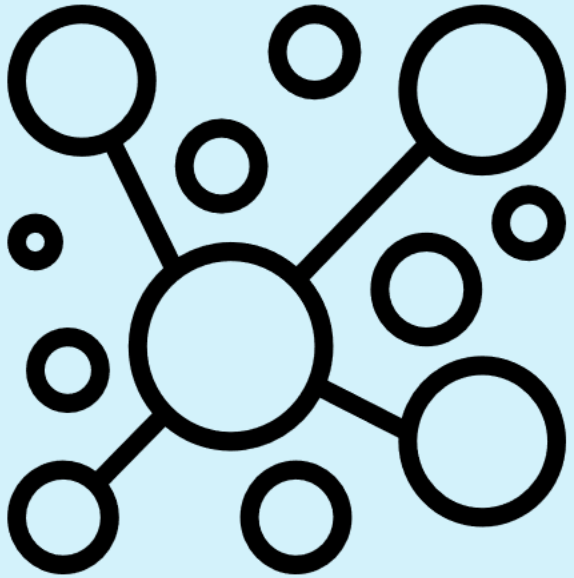

Source of Data (Producers)
CD
BLU-RAY
DVD
Receiver (Broker)
Destination of Data (Consumers)

Introduction to Apache Kafka

eazy bytes

Apache Kafka is an open-source distributed event streaming platform. It is designed to handle large-scale, real-time data streams and enables high-throughput, fault-tolerant, and scalable data processing. It is used to build real-time streaming data pipelines and applications that adapt to the data streams.

Here are some key concepts and components of Kafka:



- **Producers:** Producers are responsible for publishing messages to Kafka topics. They write messages to a specific topic, and Kafka appends these messages to the topic's log.
- **Topics:** Kafka organizes data into topics. A topic is a particular stream of data that can be divided into partitions. Each message within a topic is identified by its offset.
- **Brokers:** Brokers are the Kafka servers that manage the storage and replication of topics. They are responsible for receiving messages from producers, assigning offsets to messages, and serving messages to consumers.
- **Partitions:** Topics can be divided into multiple partitions, allowing for parallel processing and load balancing. Each partition is an ordered, immutable sequence of messages, and each message within a partition has a unique offset.
- **Offsets:** Offsets are unique identifiers assigned to each message within a partition. They are used to track the progress of consumers. Consumers can control their offsets, enabling them to rewind or skip messages based on their needs.

Introduction to Apache Kafka

eazy bytes

- **Replication:** Kafka allows topics to be replicated across multiple brokers to ensure fault tolerance. Replication provides data redundancy, allowing for failover and high availability.

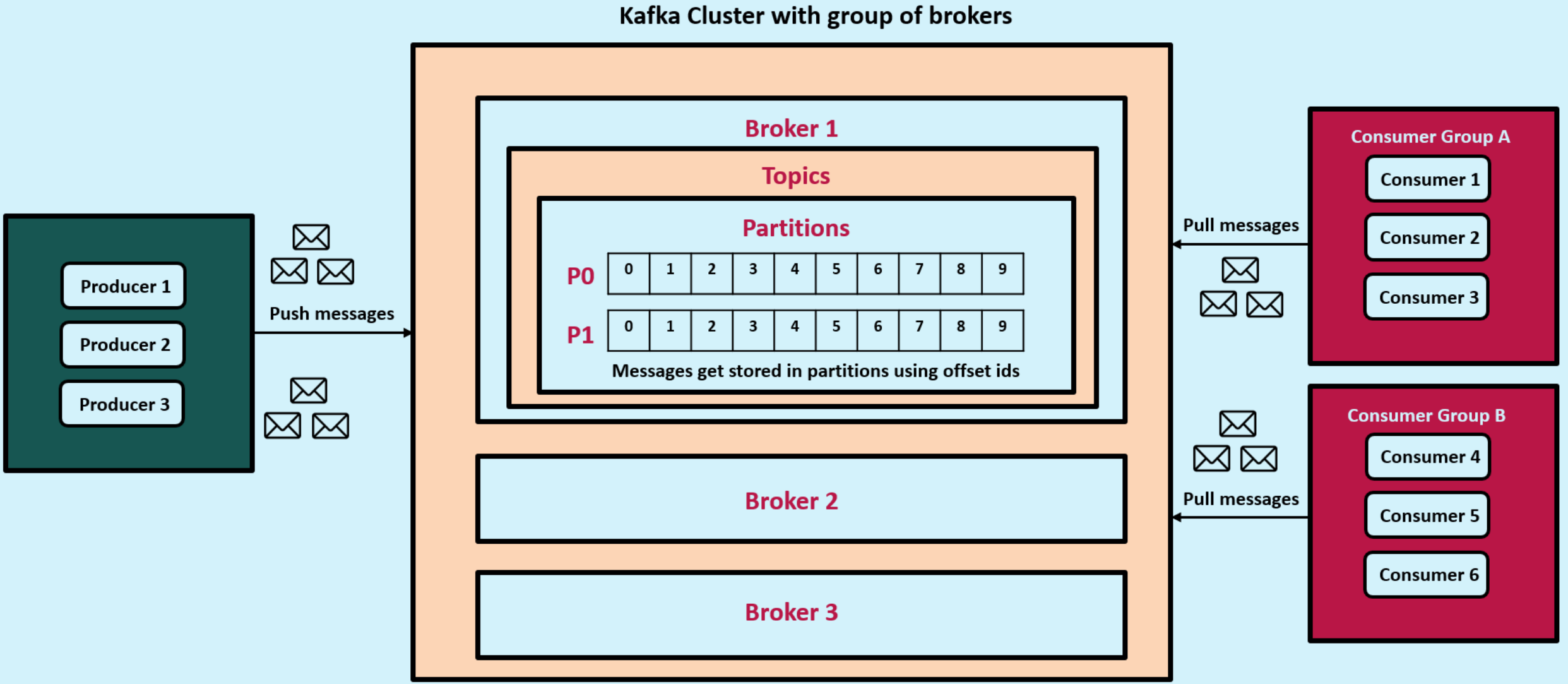
- **Consumers:** Consumers read messages from Kafka topics. They subscribe to one or more topics and consume messages by reading from specific partitions within those topics. Each consumer maintains its offset to track its progress in the topic.

- **Consumer Groups:** Consumers can be organized into consumer groups. Each message published to a topic is delivered to only one consumer within each group. This enables parallel processing of messages across multiple consumers.

- **Streams:** Kafka Streams is a client library that enables stream processing within Kafka. It allows you to build applications that consume, transform, and produce data in real-time.

Introduction to Apache Kafka

eazy bytes



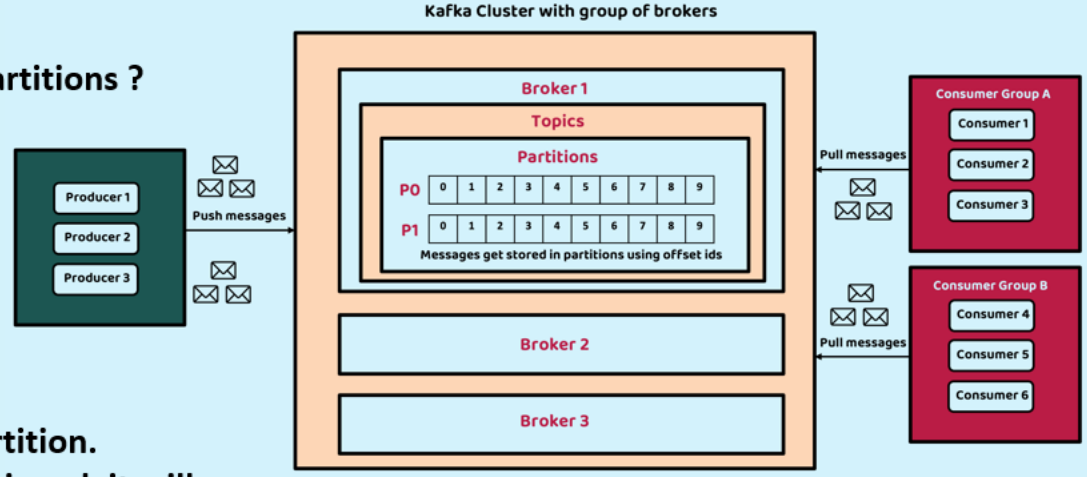
Introduction to Apache Kafka

eazy bytes

A Kafka cluster can have any number of producers, consumers, brokers. For a production set up, atleast 3 brokers is recommended. This helps in maintaining replications, fault tolerant system etc.

A Kafka broker can have any number of topics. Topic is a category under which producers can write and interested, authorized consumers can read data. For example, we can have topics like sendCommunication, dispatchOrder, purgeData etc.

Inside each topic, we can have any number of partitions. Why do we need partitions ? Since Kafka producers can handle enormous amount of data, it is not possible to store in a single server (broker). Therefore, a topic will be partitioned into multiple parts and distributed across multiple brokers, since Kafka is a distributed system. For example, we can store all customers data from a state, zipcode, region etc. inside a partition and the same can be replicated as per the configurations.



Offsets is a sequence id assigned to a message as they get stored inside a partition. The offset number starts from 0 and followed by 1,2,3.... Once offset id is assigned, it will never change. These are similar to sequence ids inside the DB tables.

By keeping track of offsets, Kafka provides reliability, fault tolerance, and flexibility to consumers. Consumers have fine-grained control over their progress, enabling them to manage message ordering, replay messages, ensure message delivery, and facilitate parallel processing.

Producer side story

eazy bytes

1. **Producer Configuration:** Before pushing a message into Kafka, a producer needs to be configured. This involves setting up properties such as the Kafka broker addresses, serialization format for messages, and other optional configurations like compression or batching.

2. **Topic Selection:** The producer needs to specify the topic to which it wants to push the message. Topics are predefined streams of data within Kafka. If the topic doesn't exist, it can be created dynamically, depending on the broker's configuration.

3. **Message Production:** The producer sends the message to Kafka by using the Kafka client library's API. The producer specifies the target topic and the serialized message. It may also provide a partition key (optional) to control which partition the message should be written to.

4. **Partition Assignment:** If a partition key is provided, Kafka uses it to determine the target partition for the message. If no partition key is provided, Kafka uses a round-robin or hashing algorithm to distribute messages evenly across partitions.

5. **Message Routing & offset assignment:** The producer sends the message to the appropriate Kafka broker based on the target topic and the partition assigned to the message. The broker receives the message and appends it to the log of the corresponding partition in a durable and ordered manner with the help of offset id.

6. **Message Replication:** Kafka ensures high availability and fault tolerance by replicating messages across multiple brokers. Once the message is written to the leader partition, Kafka asynchronously replicates it to other replicas of the partition.

7. **Acknowledgment and Error Handling:** The producer receives an acknowledgment from Kafka once the message is successfully written to the leader partition. The producer can handle any potential errors, retries, or failures based on the acknowledgment received. Depending on the acknowledgment mode configured, the producer may wait for acknowledgment from all replicas or just the leader replica.

Consumer side story

eazy bytes

1. **Consumer Group and Topic Subscription:** Consumers in Kafka are typically organized into consumer groups. Before reading messages, a consumer needs to join a consumer group and subscribe to one or more topics. This subscription specifies which topics the consumer wants to consume messages from.

2. **Partition Assignment:** Kafka assigns the partitions of the subscribed topics to the consumers within the consumer group. Each partition is consumed by only one consumer in the group. Kafka ensures a balanced distribution of partitions among consumers to achieve parallel processing.

3. **Offset Management:** Each consumer maintains its offset for each partition it consumes. Initially, the offset is set to the last committed offset or a specified starting offset. As the consumer reads messages, it updates its offset to keep track of the progress.

4. **Fetch Request:** The consumer sends fetch requests to the Kafka broker(s) it is connected to. The fetch request includes the topic, partition, and the offset from which the consumer wants to read messages. The request also specifies the maximum number of messages to be fetched in each request.

5. **Message Fetching:** Upon receiving the fetch request, the Kafka broker retrieves the requested messages from the corresponding partition's log. It sends the messages back to the consumer in a fetch response. The response contains the messages, their associated offsets, and metadata.

6. **Message Processing:** Once the consumer receives the messages, it processes them according to its application logic. This processing can involve transformations, aggregations, calculations, or any other operations based on the business requirements.

7. **Committing the Offset:** After successfully processing a batch of messages, the consumer needs to commit the offset to Kafka. This action signifies that the consumer has completed processing the messages up to that offset. Committing the offset ensures that the consumer's progress is persisted and can be resumed from that point in case of failure or restart.

8. **Polling Loop:** The consumer repeats the process of sending fetch requests, receiving messages, processing them, and committing the offset in a continuous loop. This loop allows the consumer to continuously consume and process new messages as they become available.

Steps to use Apache Kafka in the place of RabbitMQ

eazy bytes

Below are the steps to use Apache Kafka in the place of RabbitMQ,

1. **Add maven dependencies:** Add the maven dependency spring-cloud-stream-binder-kafka in the place of spring-cloud-stream-binder-rabbitmq dependency

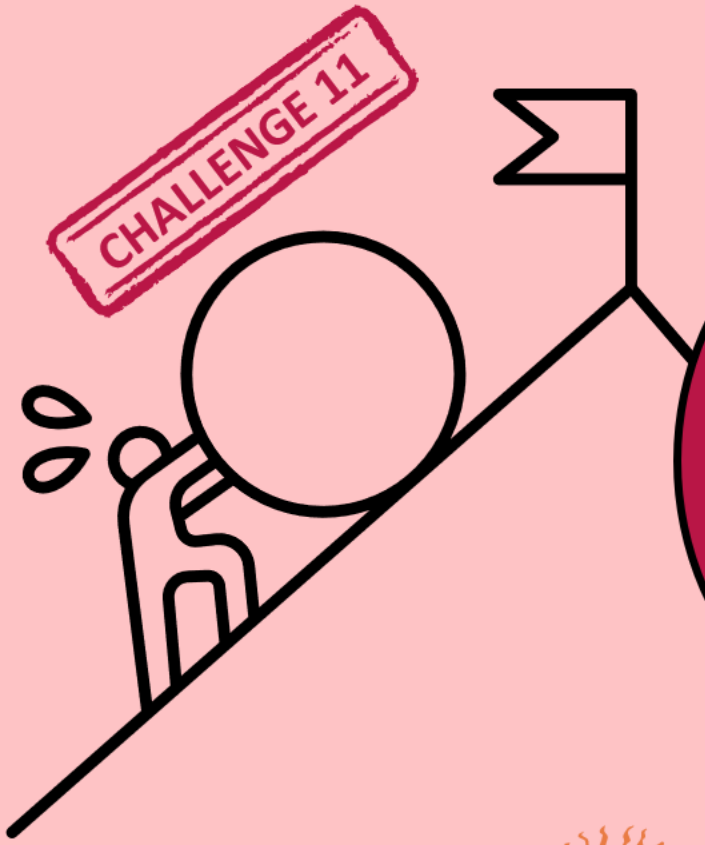
2. Add Kafka related properties inside the application.yml file of both accounts and message services

```
spring:
  application:
    name: "accounts"
  cloud:
    function:
      definition: updateCommunication
    stream:
      bindings:
        updateCommunication-in-0:
          destination: communication-sent
          group:  ${spring.application.name}
        sendCommunication-out-0:
          destination: send-communication
      kafka:
        binder:
          brokers:
            - localhost:9092
```

```
spring:
  application:
    name: message
  cloud:
    function:
      definition: email|sms
    stream:
      bindings:
        emailsms-in-0:
          destination: send-communication
          group: ${spring.application.name}
        emailsms-out-0:
          destination: communication-sent
      kafka:
        binder:
          brokers:
            - localhost:9092
```

CONTAINER ORCHESTRATION

eazy bytes



AUTOMATING THE DEPLOYMENTS, ROLLOUTS & ROLLBACKS?

How do we automate deployment of the containers into a complex cluster env and perform rollout of new versions of the containers with out down time along with an option of automatic rollback in case of any issues?

MAKING SURE OUR SERVICES ARE SELF-HEALING?

How do we automatically restarts containers that fail, replaces containers, kills containers that don't respond to your user-defined health check, and doesn't advertise them to clients until they are ready to serve.

AUTOSCALING OUR SERVICES ?

How do we monitor our services and scale them automatically based on metrics like CPU Utilization etc. ?

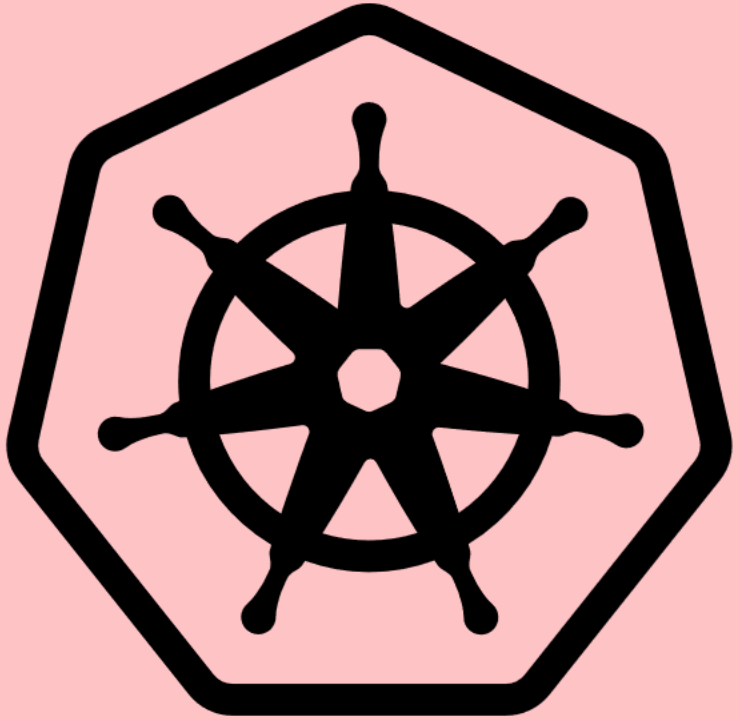
Kubernetes is an open-source container orchestration platform that automates the deployment, scaling, and management of containerized applications. It was originally developed by Google and is now maintained by the Cloud Native Computing Foundation (CNCF).

WHAT IS KUBERNETES ?

eazy bytes

Kubernetes, is an open-source system for automating deployment, scaling, and managing containerized applications. It is the most famous orchestration platform and it is cloud neutral.

Google open-sourced the Kubernetes project in 2014. Kubernetes combines over 15 years of Google's experience running production workloads at scale with best-of-breed ideas and practices from the community.



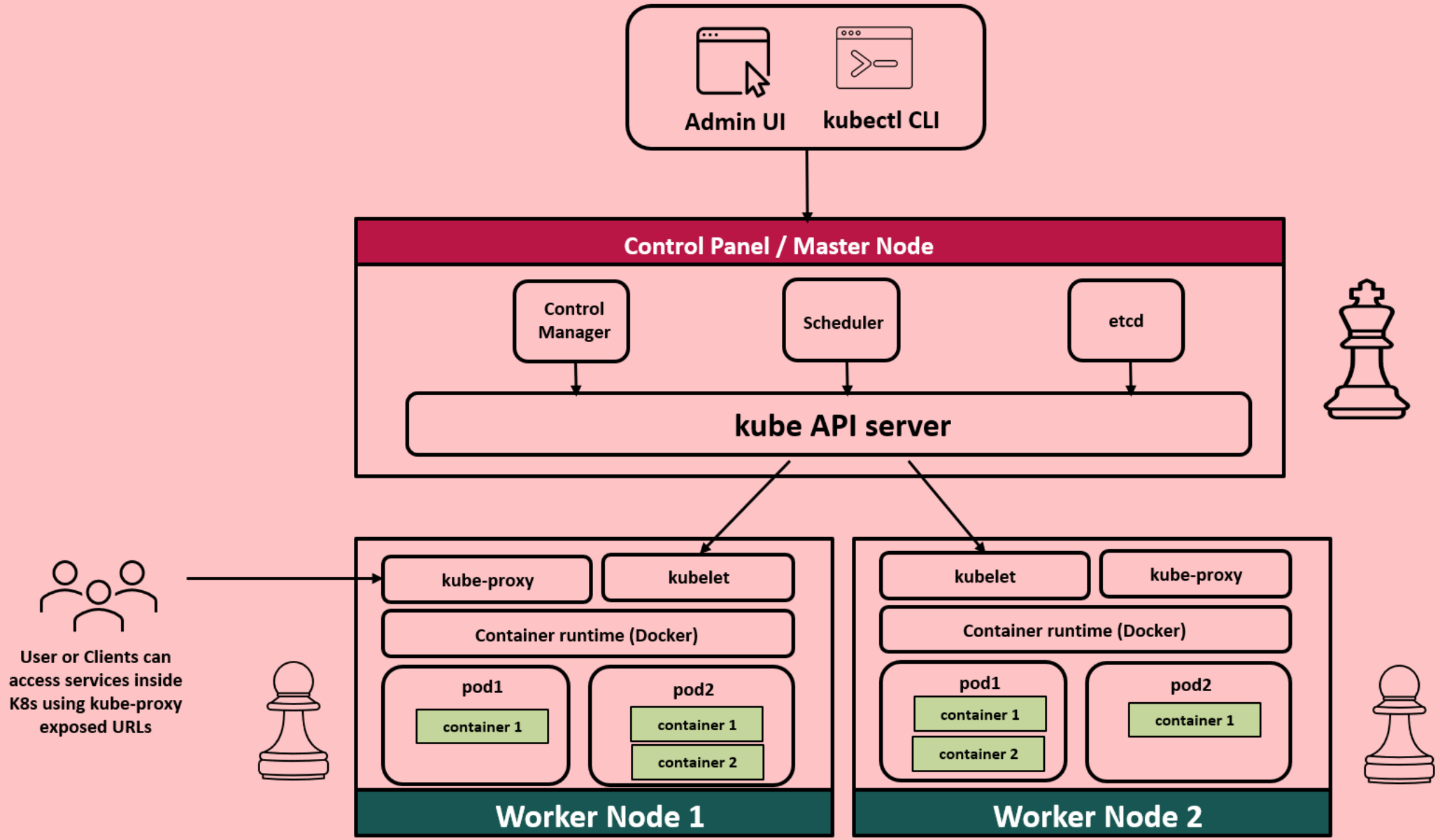
Kubernetes provides you with a framework to run distributed systems resiliently. It takes care of scaling and failover for your application, provides deployment patterns, and more. It provides you with:

- Service discovery and load balancing
- Container & storage orchestration
- Automated rollouts and rollbacks
- Self-healing
- Secret and configuration management

The name Kubernetes originates from Greek, meaning helmsman or pilot. K8s as an abbreviation results from counting the eight letters between the "K" and the "s".

KUBERNETES INTERNAL ARCHITECTURE

eazy bytes

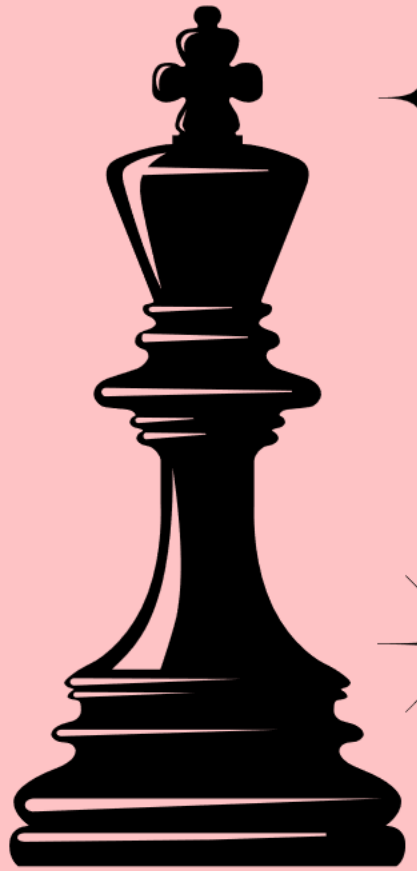


Components of Control Panel (Master Node)

eazy bytes

The master node is responsible for managing an entire cluster. It monitors the health check of all the nodes in the cluster, stores members' information regarding different nodes, plans the containers that are scheduled to certain worker nodes, monitors containers and nodes, etc. So, when a worker node fails, the master moves the workload from the failed node to another healthy worker node.

Below are the details of the four basic components present inside the control panel,



- **API server** - The API server is the primary interface for interacting with the Kubernetes cluster. It exposes the Kubernetes API, which allows users and other components to communicate with the cluster. All administrative operations and control commands are sent to the API server, which then processes and validates them.

- **Scheduler** - The scheduler is responsible for placing Pods onto available nodes in the cluster. It takes into account factors like resource requirements, affinity, anti-affinity, and other constraints to make intelligent decisions about which node to assign a Pod to. The scheduler continuously monitors the cluster and ensures that Pods are distributed optimally.

- **Controller manager** - The controller manager maintains the cluster. It handles node failures, replicates components, maintains the correct number of pods, etc. It constantly tries to keep the system in the desired state by comparing it with the current state of the system.

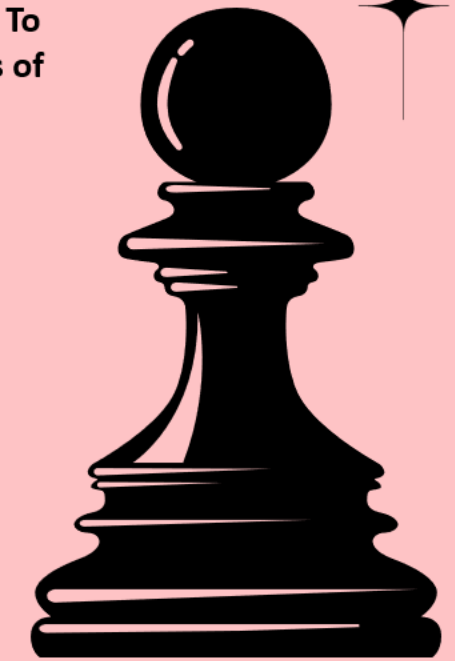
- **etcd** - etcd is a distributed key-value store that serves as the cluster's primary data store. It stores the configuration data and the desired state of the system, including information about Pods, Services, ReplicationControllers, and more. The API server interacts with etcd to read and write cluster data.

Components of Worker Node

eazy bytes

The worker node is nothing but a virtual machine (VM) running in the cloud or on-prem (a physical server running inside your data center). So, any hardware capable of running container runtime can become a worker node. These nodes expose underlying compute, storage, and networking to the applications. Worker nodes do the heavy-lifting for the application running inside the Kubernetes cluster. Together, these nodes form a cluster – a workload assign is run to them by the master node component, similar to how a manager would assign a task to a team member. This way, we will be able to achieve fault-tolerance and replication.

Pods are the smallest unit of deployment in Kubernetes just as a container is the smallest unit of deployment in Docker. To understand in an easy way, we can say that pods are nothing but lightweight VMs in the virtual world. Each pod consists of one or more containers. Each time a pod spins up, it gets a new IP address with a virtual IP range assigned by the pod networking solution.



Below are the details of the three basic components present inside the worker node,

- **Kubelet** is an agent that runs on each worker node and communicates with the control plane components. It receives instructions from the control plane, such as Pod creation and deletion requests, and ensures that the desired state of Pods is maintained on the node. The kubelet is responsible for starting, stopping, and monitoring containers based on Pod specifications.

- **Kube-proxy** is a network proxy that runs on each node in your cluster, implementing part of the Kubernetes Service concept. kube-proxy maintains network rules on nodes. These network rules allow network communication to your Pods from network sessions inside or outside of your cluster.

- **Container Runtime** is responsible for running and managing containers on a worker node. Kubernetes supports multiple container runtimes, with Docker being the most commonly used. Other runtimes like containerd and rkt are also supported. The container runtime pulls container images, creates and manages container instances, and handles container lifecycle operations.

Kubernetes manifest file to create ConfigMap

eazy bytes

A Kubernetes ConfigMap is an essential Kubernetes resource used to store configuration data separately from the application code.

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: eazybank-configmap
data:
  SPRING_PROFILES_ACTIVE: prod
```

The apiVersion and kind fields are required for all Kubernetes objects. When creating a config map inside K8s, the kind should be "ConfigMap"

The metadata field contains the name of the ConfigMap and other metadata about the object.

The data field is where the key-value pairs are stored. The keys can be any alphanumeric string, and the values can be strings, numbers, or binary data.

Kubernetes manifest file to deploy a Container

eazy bytes

In Kubernetes, a Deployment is a high-level resource used to manage the deployment and scaling of containerized applications. It provides a declarative way to define and maintain the desired state of your application. When you create a Deployment, Kubernetes ensures that the specified number of replicas of your application are running and automatically handles scaling, rolling updates, and rollbacks.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: accounts-deployment
  labels:
    app: accounts
spec:
  replicas: 1
  selector:
    matchLabels:
      app: accounts
  template:
    metadata:
      labels:
        app: accounts
    spec:
      containers:
      - name: accounts
        image: eazybytes/accounts:latest
        ports:
        - containerPort: 8080
        env:
        - name: SPRING_PROFILES_ACTIVE
          valueFrom:
            configMapKeyRef:
              name: eazybank-configmap
              key: SPRING_PROFILES_ACTIVE
```

The apiVersion and kind fields are required for all Kubernetes objects. For deployment manifest file, the kind should be “Deployment”

Metadata: The metadata section contains information about the Deployment, such as its name and labels. The Deployment is named "accounts-deployment", and it has the label "app: accounts".

Spec: The spec section defines the desired state of the Deployment.

Replicas: The replicas field is set to 1, indicating that only one replica of the container should be running at any given time.

Selector: The selector field is used to select the pods controlled by this Deployment. In this case, it's using the label "app: accounts" to identify the pods.

Kubernetes manifest file to deploy a Container

eazy bytes

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: accounts-deployment
  labels:
    app: accounts
spec:
  replicas: 1
  selector:
    matchLabels:
      app: accounts
  template:
    metadata:
      labels:
        app: accounts
    spec:
      containers:
      - name: accounts
        image: eazybytes/accounts:latest
        ports:
        - containerPort: 8080
        env:
        - name: SPRING_PROFILES_ACTIVE
          valueFrom:
            configMapKeyRef:
              name: eazybank-configmap
              key: SPRING_PROFILES_ACTIVE
```

Template: The template section specifies the pod template that will be used to create pods for this Deployment.

Metadata: The metadata section inside the template defines the labels for the pods. The pod will have the label "app: accounts".

Spec: The spec section inside the template specifies the details of the pod's specification.

Containers: The containers field lists the containers that should be part of the pod. In this case, there is one container named "accounts" based on the "eazybytes/accounts:latest" image.

Ports: The ports section exposes port 8080 of the container.

Environment Variables: The env section sets an environment variable named "SPRING_PROFILES_ACTIVE" and assigns it a value from a ConfigMap.

valueFrom: The valueFrom field allows you to reference data from a ConfigMap. In this case, it is using configMapKeyRef to fetch the value of the "SPRING_PROFILES_ACTIVE" key from the ConfigMap named "eazybank-configmap".

Kubernetes manifest file to create a service

eazy bytes

In Kubernetes, a Service is an essential resource that provides network connectivity to a set of pods. It acts as a stable endpoint for accessing and load balancing traffic across multiple replicas of a pod. Services abstract away the underlying network details, allowing pods to be more dynamic and scalable without affecting how clients access them.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: accounts-service
spec:
  selector:
    app: accounts
  type: ClusterIP
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 8080
```

The apiVersion and kind fields are required for all Kubernetes objects. For service manifest file, the kind should be “Service”

Metadata: The metadata section contains information about the Service, such as its name.

Spec: The spec section defines the desired state of the Service.

Selector: The selector field is used to select the pods that the Service will route traffic to. In this case, it uses the label "app: accounts" to select the pods controlled by the Deployment with the same label.

Type: The type field specifies the type of Service. In this case, it is set to "ClusterIP," which means that the Service will be accessible only from within the cluster.

Kubernetes manifest file to create a service

eazy bytes

```yaml
apiVersion: v1
kind: Service
metadata:
  name: accounts-service
spec:
  selector:
    app: accounts
  type: ClusterIP
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 8080
```

Ports: The ports section defines the ports that the Service should listen on and forward traffic to.

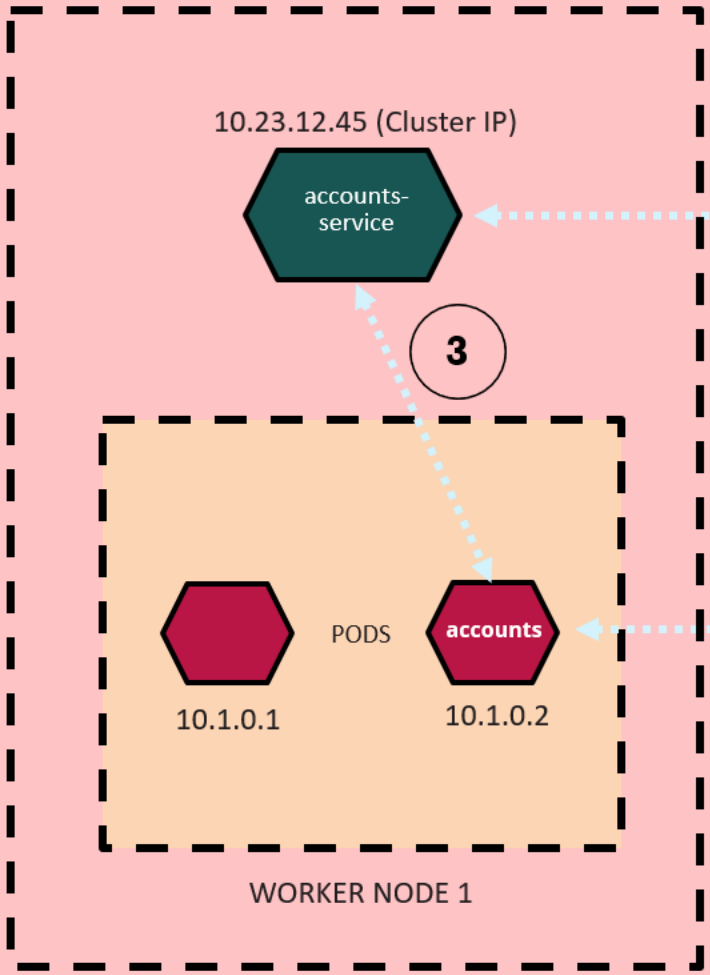
protocol: The protocol field specifies the protocol used for the service port. In this case, it's TCP.

port: The port field is the port number on which the Service will listen for incoming traffic.

targetPort: The targetPort field is the port number on the pods to which the incoming traffic will be forwarded.

How K8s deployment & services tied together

eazy bytes



2

1

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: accounts-deployment
  labels:
    app: accounts
spec:
  replicas: 1
  selector:
    matchLabels:
      app: accounts
  template:
    metadata:
      labels:
        app: accounts
    spec:
      containers:
      - name: accounts
        image: eazybytes/accounts:latest
        ports:
        - containerPort: 8080
```

3

```yaml
apiVersion: v1
kind: Service
metadata:
  name: accounts-service
spec:
  selector:
    app: accounts
  type: ClusterIP
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 8080
```

1. K8s Deployment manifest will give instructions to deploy the containers into a pod inside one of the worker node of the K8s cluster
2. K8s Service manifest will give instructions to create a service inside the K8s cluster which tracks service registration & discovery of a specific service/deployment
3. The binding between a container running inside a Pod and a service will be done based “app” label & selector inside the manifest files.

Kubernetes Service Types

eazy bytes

Below 3 Kubernetes Service types are used majorly inside the K8s clusters

ClusterIP Service

This is the default service that uses an internal Cluster IP to expose Pods. In ClusterIP, the services are not available for external access of the cluster and used for internal communications between different Pods or microservices in the cluster.

NodePort Service

This service exposes outside and allows the outside traffic to connect to K8s Pods through the node port which is the port opened at Node end. The Pods can be accessed from external using <NodeIp>:<Nodeport>

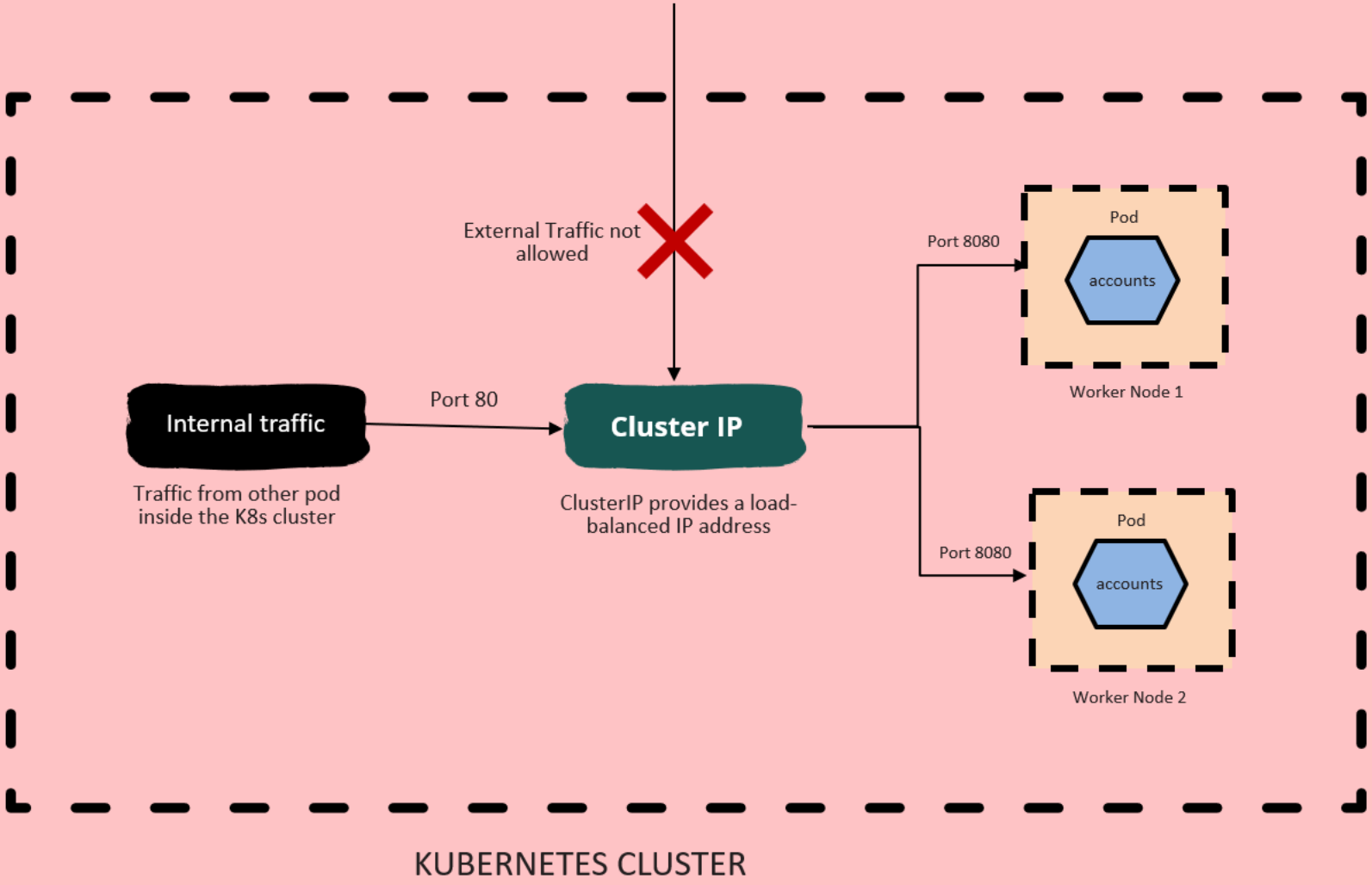
LoadBalancer Service

This service is exposed like in NodePort but creates a load balancer in the cloud where K8s is running that receives external requests to the service. It then distributes them among the cluster nodes using NodePort.

K8s CLUSTER IP SERVICE

eazy bytes

ClusterIP service creates an internal IP address for use within the K8s cluster.
Good for internal-only applications that support other workloads within the cluster.



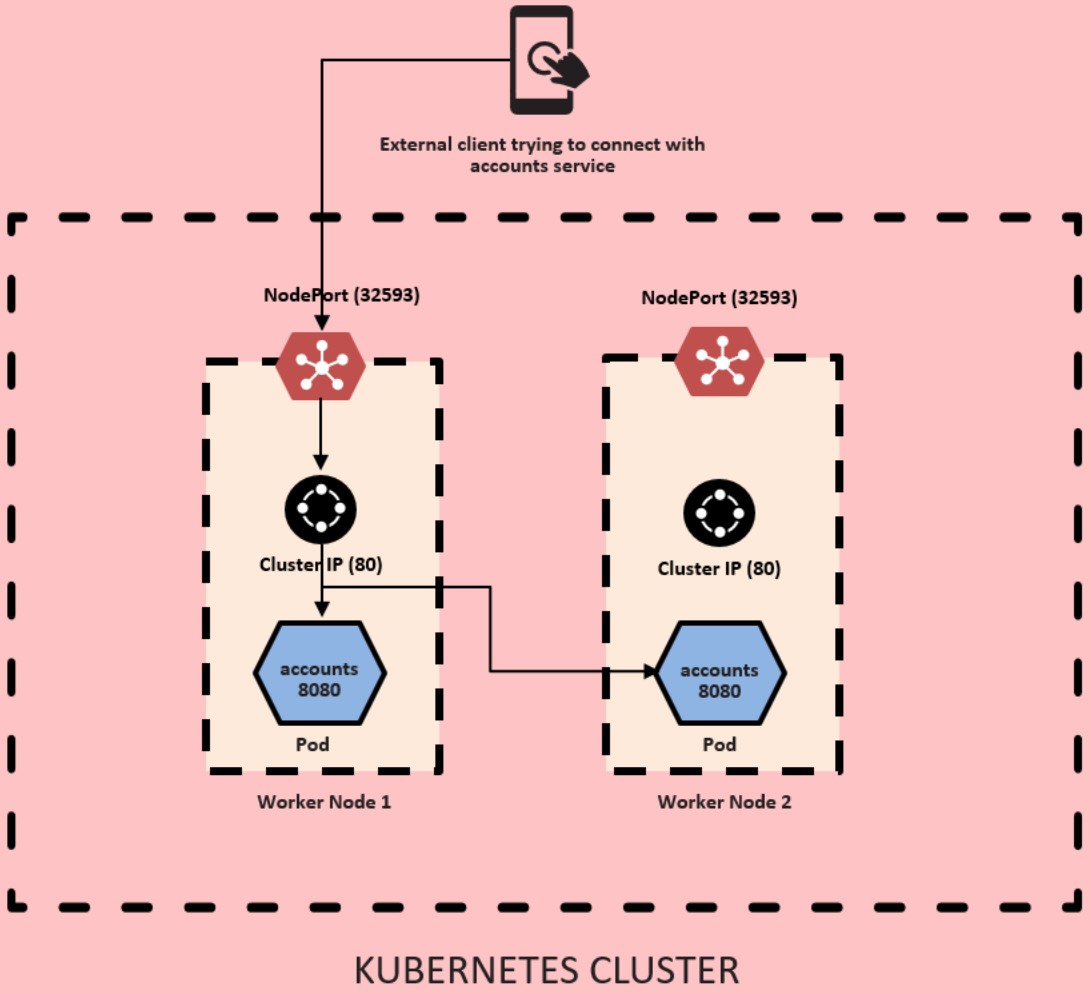

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: accounts-deployment
  labels:
    app: accounts
spec:
  replicas: 2
  selector:
    matchLabels:
      app: accounts
  template:
    metadata:
      labels:
        app: accounts
    spec:
      containers:
      - name: accounts
        image: eazybytes/accounts:latest
        ports:
        - containerPort: 8080
```

```yaml
apiVersion: v1
kind: Service
metadata:
  name: accounts-service
spec:
  selector:
    app: accounts
  type: ClusterIP
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
```

K8s NODEPORT SERVICE

eazy bytes

Services of type NodePort build on top of ClusterIP type services by exposing the ClusterIP service outside of the cluster on high ports (default 30000-32767). If no port number is specified then Kubernetes automatically selects a free port. The local kube-proxy is responsible for listening to the port on the node and forwarding client traffic on the NodePort to the ClusterIP.



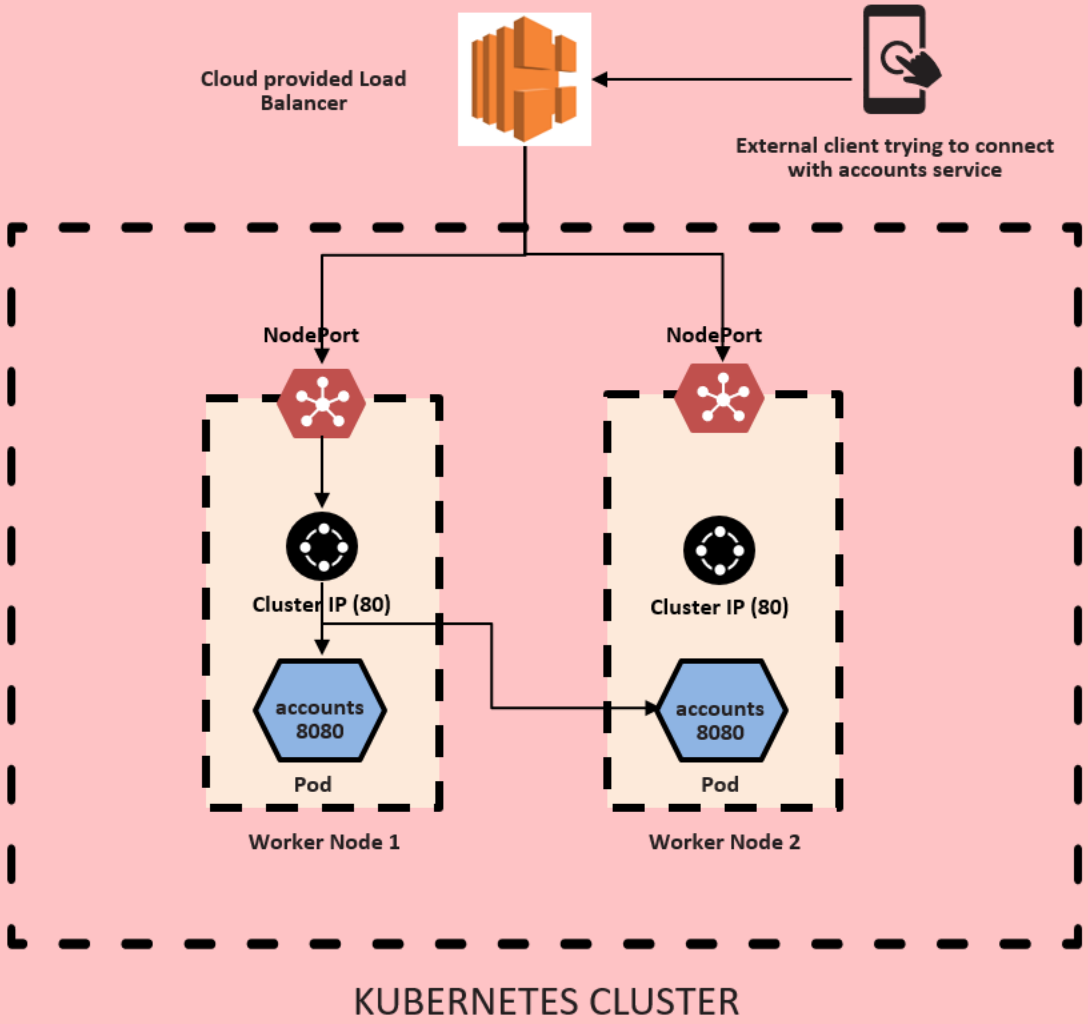

```yaml
apiVersion: v1
kind: Service
metadata:
  name: accounts-service
spec:
  selector:
    app: accounts
  type: NodePort
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
      nodePort: 32593
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: accounts-deployment
  labels:
    app: accounts
spec:
  replicas: 2
  selector:
    matchLabels:
      app: accounts
  template:
    metadata:
      labels:
        app: accounts
    spec:
      containers:
      - name: accounts
        image: eazybytes/accounts:latest
        ports:
        - containerPort: 8080
```

K8s LOADBALANCER SERVICE

eazy bytes

The LoadBalancer service type is built on top of NodePort service types by provisioning and configuring external load balancers from public and private cloud providers. It exposes services that are running in the cluster by forwarding layer 4 traffic to worker nodes. This is a dynamic way of implementing a case that involves external load balancers and NodePort type services.




```yaml
apiVersion: v1
kind: Service
metadata:
  name: accounts-service
spec:
  selector:
    app: accounts
  type: LoadBalancer
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: accounts-deployment
  labels:
    app: accounts
spec:
  replicas: 2
  selector:
    matchLabels:
      app: accounts
  template:
    metadata:
      labels:
        app: accounts
    spec:
      containers:
      - name: accounts
        image: eazybytes/accounts:latest
        ports:
        - containerPort: 8080
```

Introduction to Helm

eazy bytes

What is HELM ?

Helm is renowned as the "package manager of Kubernetes," aiming to enhance the management of Kubernetes projects by offering users a more efficient approach to handling the multitude of YAML files involved.

With out Helm we need to maintain all K8s manifest files for Deployment, Service, ConfigMap etc. for each microservice

With out Helm, the Devops team members has to manually apply or delete all the Kubernetes YAML manifest files using kubectl

The path Helm tooks to solve this issues is by using a packaging format called charts. A chart is a collection of files that describe a related set of Kubernetes resources.

A single Helm chart might be used to deploy a simple app or something complex like a full web app stack with HTTP servers, databases, caches, and so on.

A chart can have child charts and dependent charts as well. This means that Helm can install whole dependency tree of a project with just a single command.

Problems that Helm solves

eazy bytes

With out Helm, we need to maintain the separate YAML files/manifest of K8s for all microservices inside a project like below. But majority content inside them looks similar except few dynamic values.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: accounts-service
spec:
  selector:
    app: accounts
  type: LoadBalancer
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 8080
```

accounts service manifest

```yaml
apiVersion: v1
kind: Service
metadata:
  name: loans-service
spec:
  selector:
    app: loans
  type: LoadBalancer
  ports:
    - protocol: TCP
      port: 8090
      targetPort: 8090
```

loans service manifest

```yaml
apiVersion: v1
kind: Service
metadata:
  name: cards-service
spec:
  selector:
    app: cards
  type: LoadBalancer
  ports:
    - protocol: TCP
      port: 9000
      targetPort: 9000
```

cards service manifest

Problems that Helm solves

eazy bytes

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

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

Helm service template

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

values.yaml

Problems that Helm solves

eazy bytes

HELM SUPPORTS PACKAGING OF YAML FILES

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

HELM SUPPORTS EASIER INSTALLATION

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

HELM SUPPORTS RELEASE/VERSION MANAGEMENT

Helm automatically maintains the version history of the installed manifests. Due to that rollback of entire K8s cluster to the previous working state is just a single command away.

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

Helm chart structure

eazy bytes

```
wordpress/
├──Chart.yml
├──values.yml
├──charts/
├──templates/
```

Top level wordpress folder is the name of the chart that we have given while installing/creating the chart.

Chart.yaml will have meta info about the helm chart
values.yaml will have dynamic values for the chart

charts folder will have other charts which the current chart is dependent on.

templates folder contains the manifest template yaml files

Important Helm commands

eazy bytes

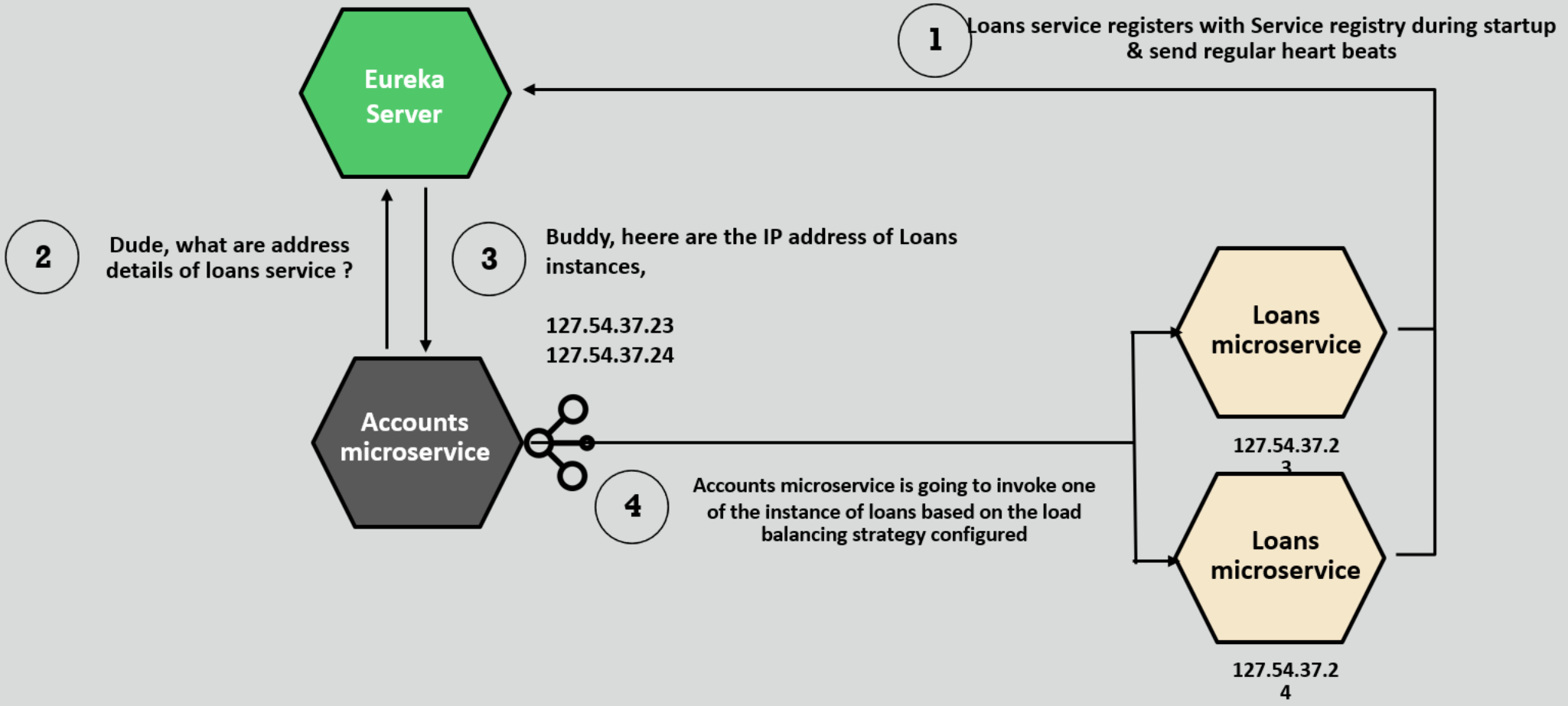
Helm commands

Command	Description
helm create eazybank	Create a blank chart with the name eazybank. Inside the eazybank folder, we can see Charts.yaml, values.yml, Charts folder, templates folder etc.
helm dependencies build	Build is used to reconstruct a chart's dependencies
helm install [NAME] [CHART]	Install the manifests mentioned in the [CHART] with a given release name inside [NAME]
helm upgrade [NAME] [CHART]	Upgrades a specified release to a new version of a chart
helm history [NAME]	Prints historical revisions for a given release.
helm rollback [NAME] [REVISION]	Roll back a release to a previous revision. The first argument of the rollback command is the name of a release, and the second is a revision (version) number. If this argument is omitted, it will roll back to the previous release.
helm uninstall [NAME]	Removes all of the resources associated with the last release of the chart as well as the release history
helm template [NAME] [CHART]	Render chart templates locally along with the values and display the output.
helm ls	This command lists all of the releases for a specified namespace

Client-side service discovery and load balancing

eazy bytes

In client-side service discovery, applications are responsible for registering themselves with a service registry like Eureka during startup and unregistering when shutting down. When an application needs to communicate with a backing service, it queries the service registry for the associated IP address. If multiple instances of the service are available, the registry returns a list of IP addresses. The client application then selects one based on its own defined load-balancing strategy. Below figure illustrates the workflow of this process

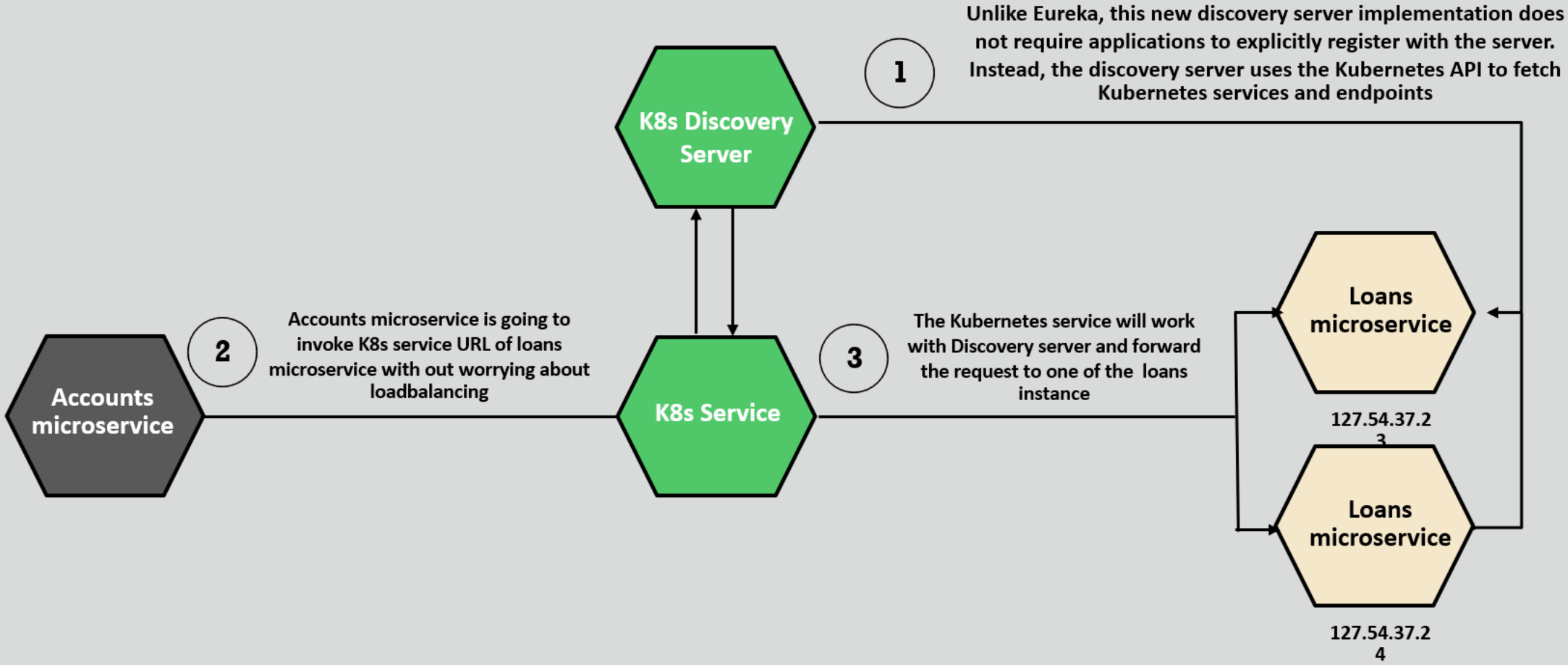


Server-side service discovery and load balancing

eazy bytes

In server-side service discovery, the K8s discovery server is responsible to monitor the application instances and maintaining the details of them. When a microservice needs to communicate with a backing service, it simply invokes the service URL exposed by the K8s. The K8s will take care of loadbalancing the requests at the server layer. So clients are free from load balancing in this scenario.

Below figure illustrates the workflow of this process,



Kubernetes support by Cloud providers

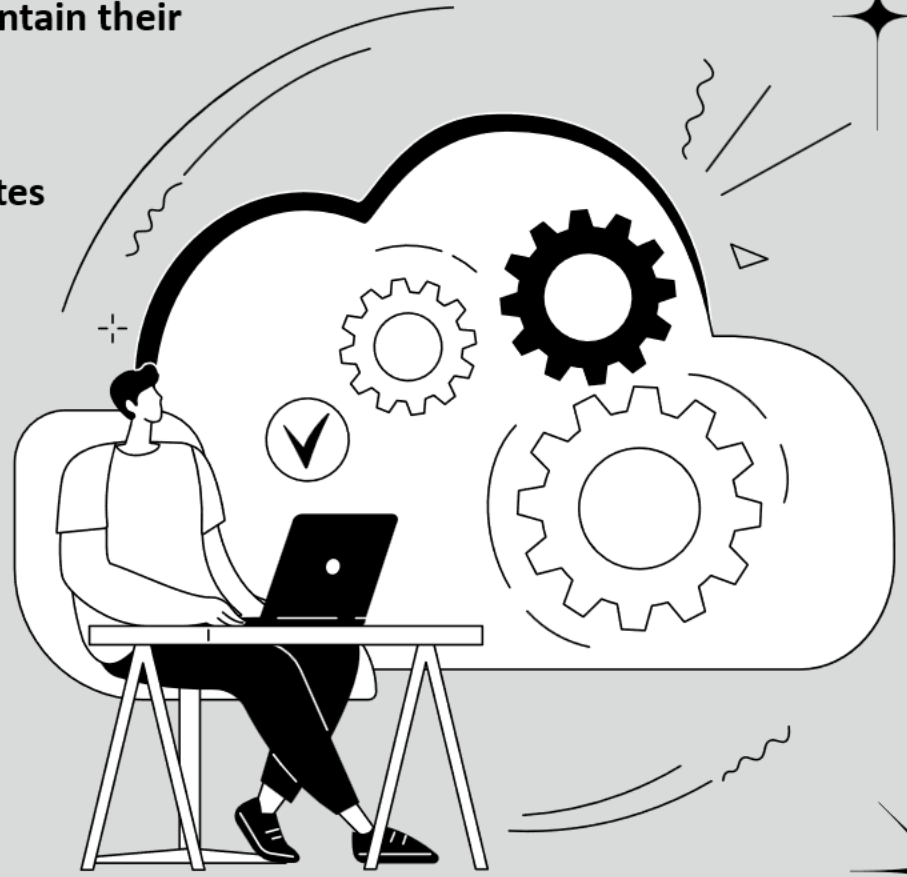
eazy bytes

Kubernetes is so modular, flexible, and extensible that it can be deployed on-prem, in a third-party data center, in any of the popular cloud providers and even across multiple cloud providers.

Creating and maintaining a K8S cluster can be very challenge in on-prem. Due that many enterprises look for the cloud providers which will make their job easy to maintain their microservice architecture using Kubernetes.

Below are the different famous cloud providers and their support to Kubernetes with the different names,

- GCP - GKE (Google Kubernetes Engine)
- AWS - EKS (Elastic Kubernetes Service)
- Azure - AKS (Azure Kubernetes Service)



Kubernetes Ingress

eazy bytes

What is Ingress ?

Ingress exposes HTTP and HTTPS routes from outside the cluster to services within the cluster. Traffic routing is controlled by rules defined on the Ingress resource. An Ingress may be configured to give Services externally-reachable URLs, load balance traffic, terminate SSL / TLS, and offer name-based virtual hosting.

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: example-ingress
spec:
  rules:
    - host: myapp.example.com
      http:
        paths:
          - path: /app1
            pathType: Prefix
            backend:
              service:
                name: app1-service
                port:
                  number: 80
          - path: /app2
            pathType: Prefix
            backend:
              service:
                name: app2-service
                port:
                  number: 80
```

What is Ingress Controller ?

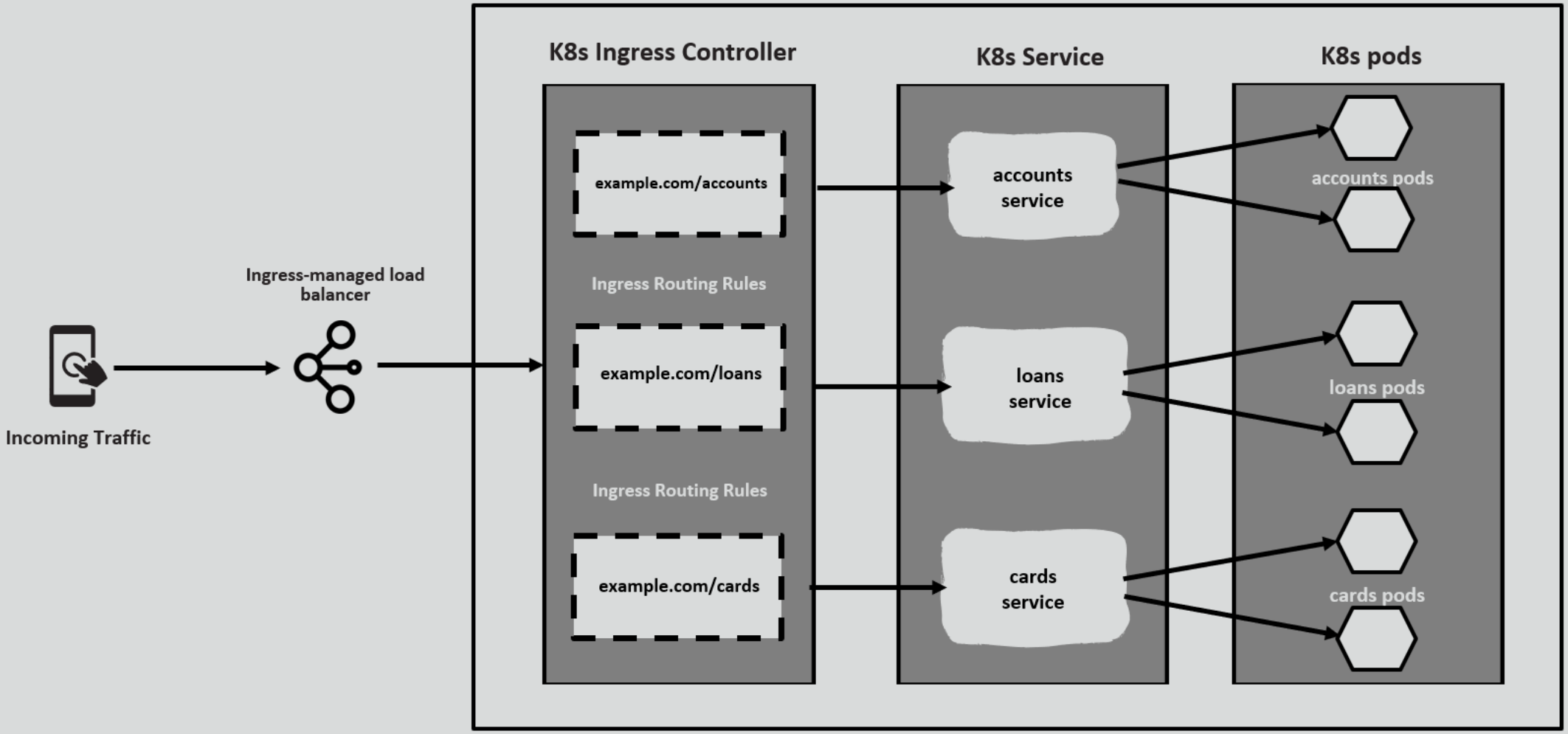
On its own, the Ingress resource doesn't do anything. You need to have an Ingress controller installed and configured in your cluster to make Ingress resources functional. Popular Ingress controllers include Nginx Ingress, Traefik, and HAProxy Ingress. The controller watches for Ingress resources and configures the underlying networking components accordingly.

Full list of Ingress Controllers available in the below URL,

https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/

Kubernetes Ingress

eazy bytes



Traffic flow in K8s Cluster with Ingress controller

Kubernetes Ingress

eazy bytes

Why Use Ingress ?

Ingress provides several benefits:

Single Entry Point: It allows you to configure a single entry point for multiple services, making it easier to manage external access to your applications.

TLS/SSL Termination: Ingress can handle TLS/SSL termination, allowing you to secure your applications with SSL/TLS certificates.

Path-Based Routing: You can route traffic to different services based on the request path (e.g., /app1 goes to one service, /app2 to another).

Host-Based Routing: You can route traffic based on the requested host or domain name (e.g., app1.example.com goes to one service, app2.example.com to another).

Load Balancing: It provides built-in load balancing for distributing traffic among multiple pods of the same service.

Annotations:
Ingress resources can be customized using annotations. Annotations allow you to configure additional settings, such as rewrite rules, custom headers, and authentication.

Kubernetes Ingress

eazy bytes

Ingress Controllers vs. Service Type LoadBalancer

Ingress controllers are often compared to using Kubernetes Service resources of type LoadBalancer. While both can expose services externally, Ingress offers more advanced routing and traffic management capabilities.

Types of traffic handled by Ingress Controller

Ingress traffic: Traffic entering a Kubernetes cluster

Egress traffic: Traffic exiting a Kubernetes cluster

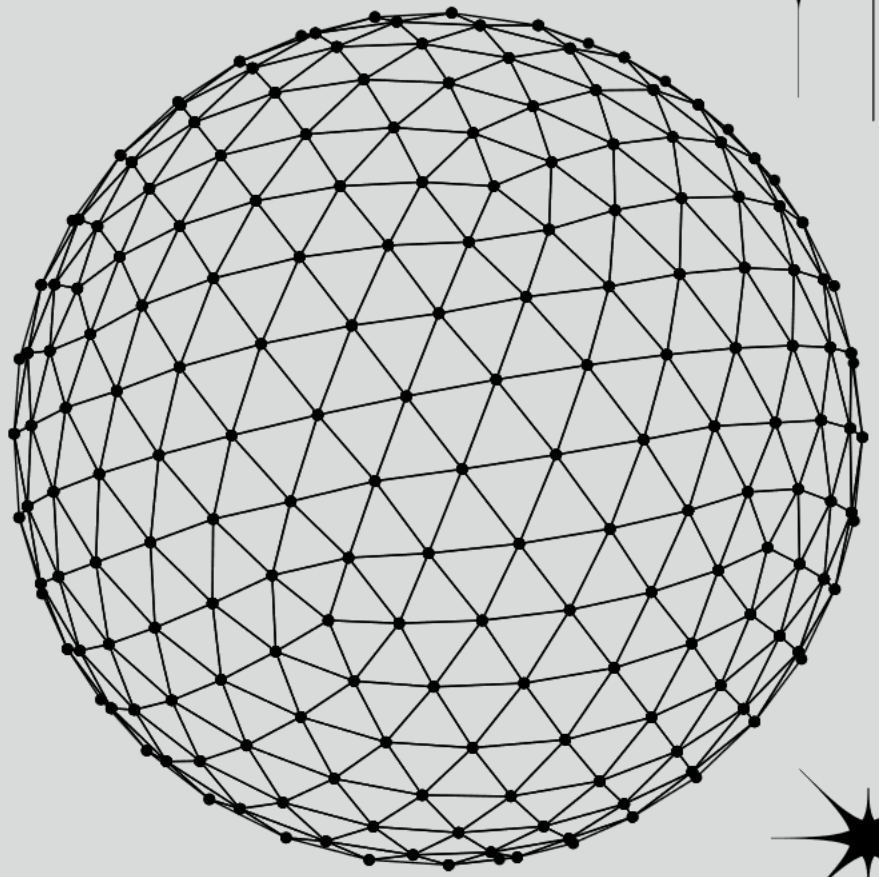
North-south traffic: Traffic entering and exiting a Kubernetes cluster (also called ingress-egress traffic)

Who handles service-service traffic ?

eazy bytes

service-to-service traffic – Traffic moving among services within a Kubernetes cluster (also called as east-west traffic). Service mesh can handle east-west traffic efficiently.

A **service mesh** is a dedicated infrastructure layer for managing communication between microservices in a containerized application. It provides a set of networking capabilities that help facilitate secure, reliable, and observable communication between services within a distributed system.



The service mesh can provide a variety of features, such as:

Service discovery: This allows services to find each other without having to hard-code the addresses in their code.

Load balancing: This ensures that traffic is distributed evenly across all instances of a service.

Circuit breaking: This prevents a service from becoming overloaded by traffic.

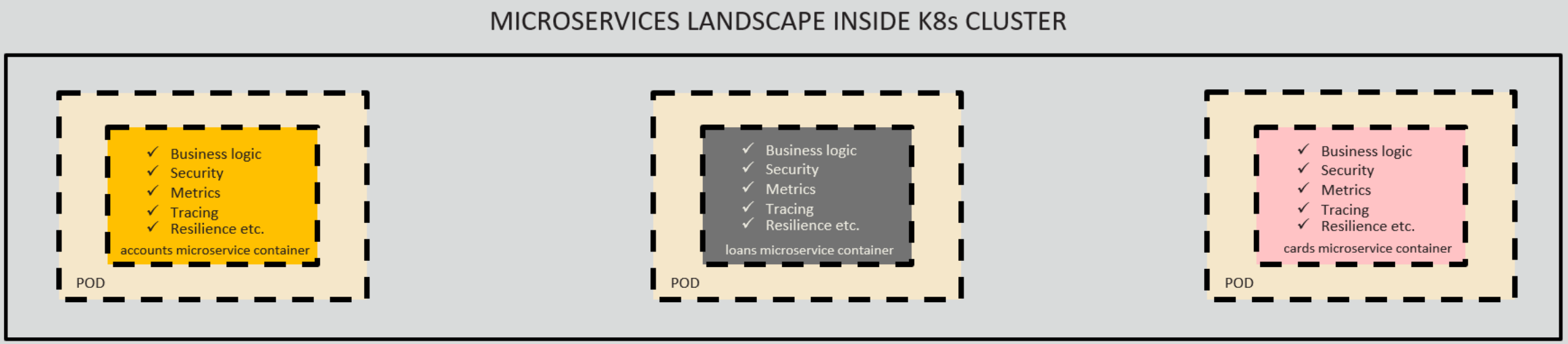
Fault tolerance: This ensures that services can continue to function even if some of their dependencies fail.

Metrics and tracing: This provides visibility into the traffic flowing through the service mesh.

Security: Service mesh can secure internal service-to-service communication in a cluster with Mutual TLS (mTLS)

Problem that Service Mesh trying to solve

eazy bytes



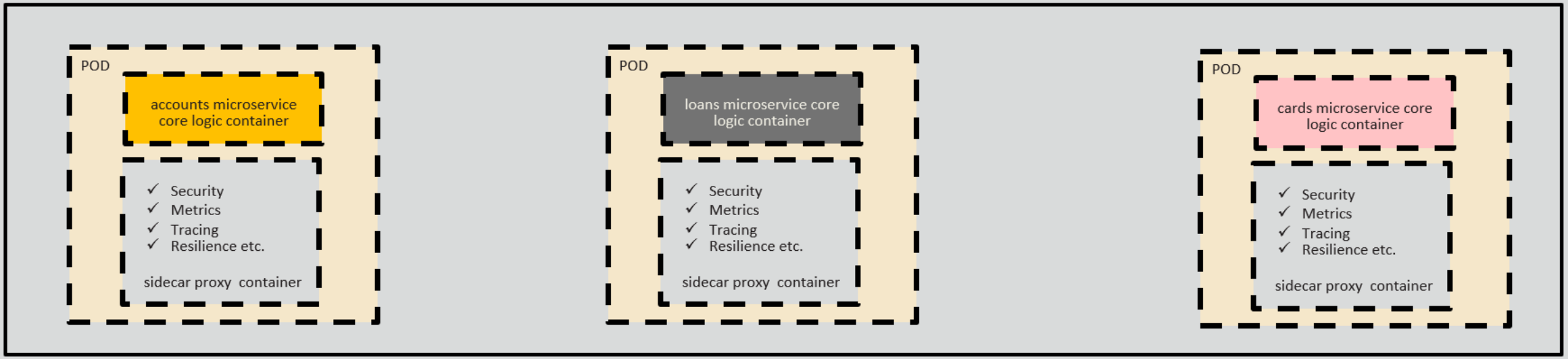
Each microservice has lot of code/configurations related to Security, tracing etc. which is not related to business logic. All the extra code/configurations that is present inside each microservice will make them complex.

Developers needs to manage these changes consistently in all the microservices which deviates from their main focus of building the business logic.

How Service Mesh makes our life easy while building microservices

eazy bytes

MICROSERVICES LANDSCAPE INSIDE K8s CLUSTER



A sidecar proxy container is deployed along with each service that you start in your cluster. This is also called as Sidecar pattern. This pattern is named Sidecar because it resembles a sidecar attached to a motorcycle. In the pattern, the sidecar is attached to a parent application and provides supporting features for the application. The sidecar also shares the same lifecycle as the parent application, being created and retired alongside the parent.

A sidecar is independent from its primary application in terms of runtime environment and programming language, so you don't need to develop one sidecar per language. Developers will also relived from developing all these non business logic related components.

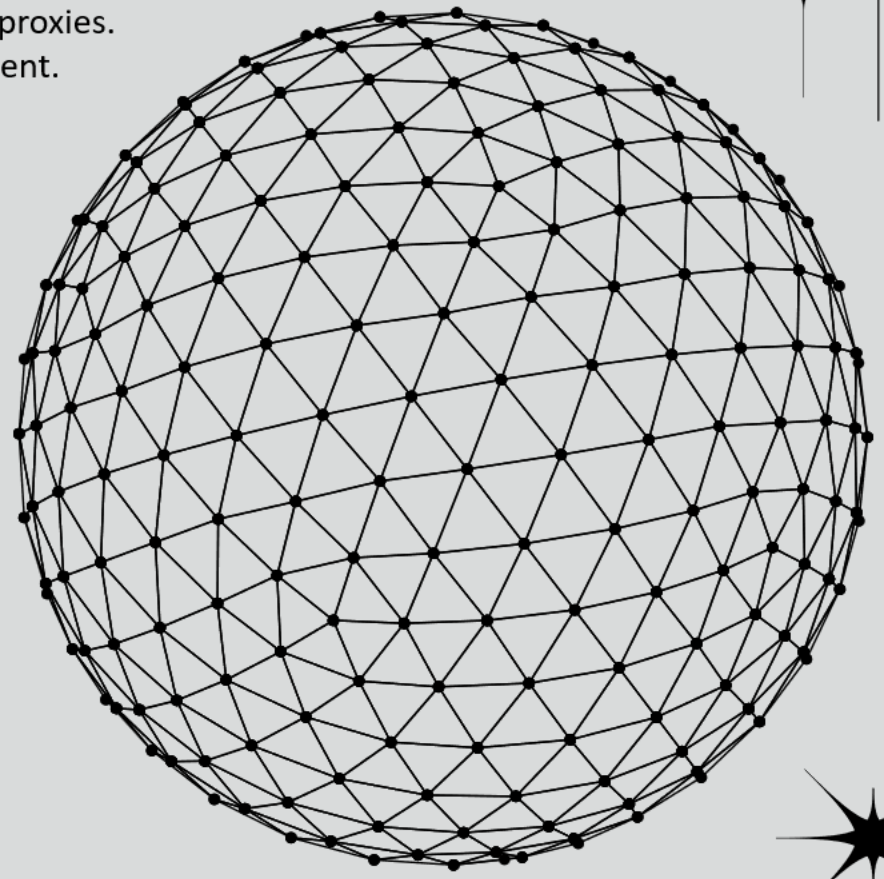
Service mesh components

eazy bytes

A service mesh typically consists of two components:

Data plane: This is responsible for routing traffic between services. It can be implemented using proxies. Each microservice instance is accompanied by a lightweight proxy (e.g., Envoy, Linkerd proxy) known as a sidecar. These proxies handle traffic to and from the service, intercepting requests and responses.

Control plane: The control plane is responsible for configuring, managing, and monitoring the proxies. It includes components like a control plane API, service discovery, and configuration management.



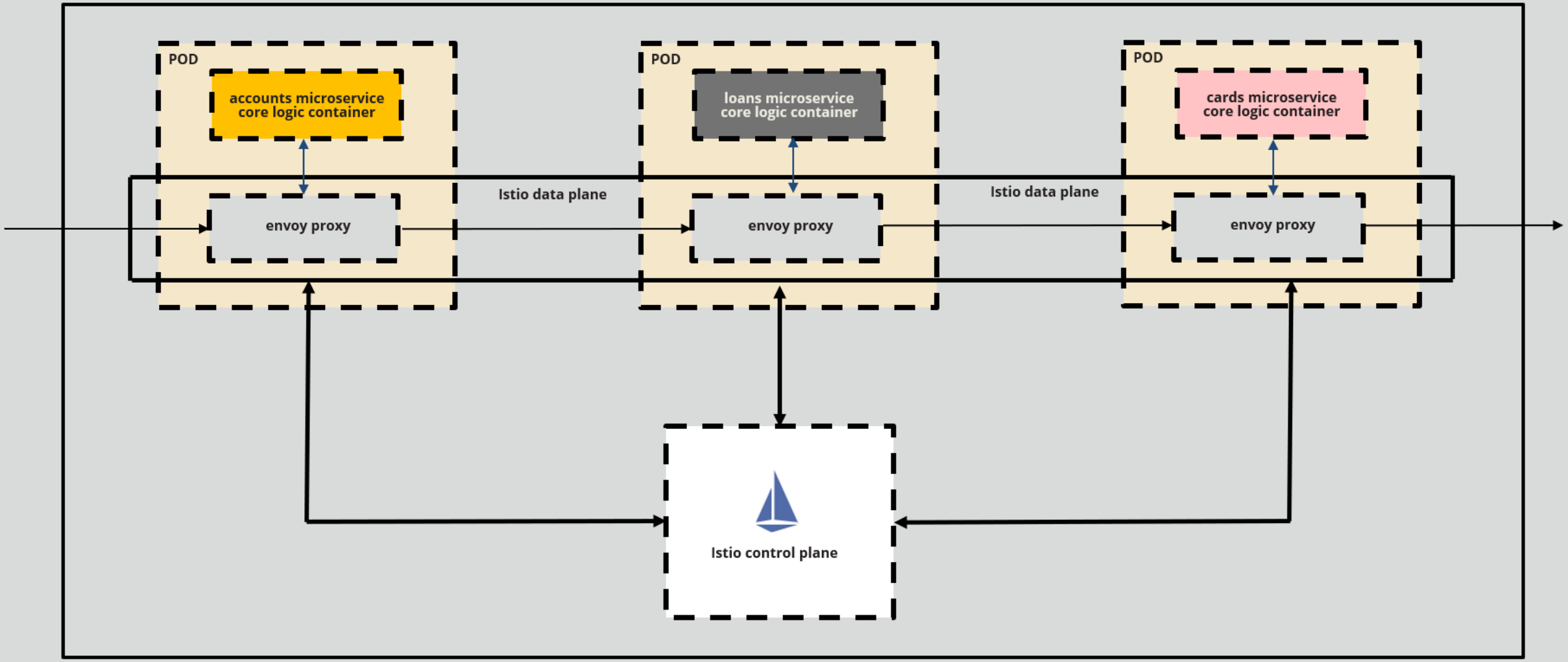
Here are some of the popular service meshes:

- ✓ Istio
- ✓ Linkerd
- ✓ Consul
- ✓ Kong
- ✓ AWS App Mesh
- ✓ Azure Service Mesh

The best service mesh for an Organization will depend on it's specific needs and requirements.

Istio service mesh components

eazy bytes



Introduction to mutual TLS (mTLS)

eazy bytes

Mutual TLS (mTLS) represents a variant of transport layer security (TLS). TLS, which succeeded secure sockets layer (SSL), stands as the prevailing standard for secure communication, prominently used in HTTPS. It facilitates secure communication that guarantees both confidentiality (protection against eavesdropping) and authenticity (protection against tampering) between a server, which needs to verify its identity to clients.

However, in scenarios where both sides require mutual identity verification, such as interactions between microservices within a Kubernetes application, traditional TLS falls short. mTLS comes into play when both parties must mutually authenticate themselves. It enhances the security provided by TLS by introducing mutual authentication between the two parties.

mTLS is often used in a Zero Trust security framework* to verify users, devices, and servers within an organization. It can also help keep APIs secure.

*Zero Trust means that no user, device, or network traffic is trusted by default, an approach that helps eliminate many security vulnerabilities.

What is TLS?

Transport Layer Security (TLS) is an encryption protocol in wide use on the Internet. TLS, which was formerly called SSL, authenticates the server in a client-server connection and encrypts communications between client and server so that external parties cannot spy on the communications.

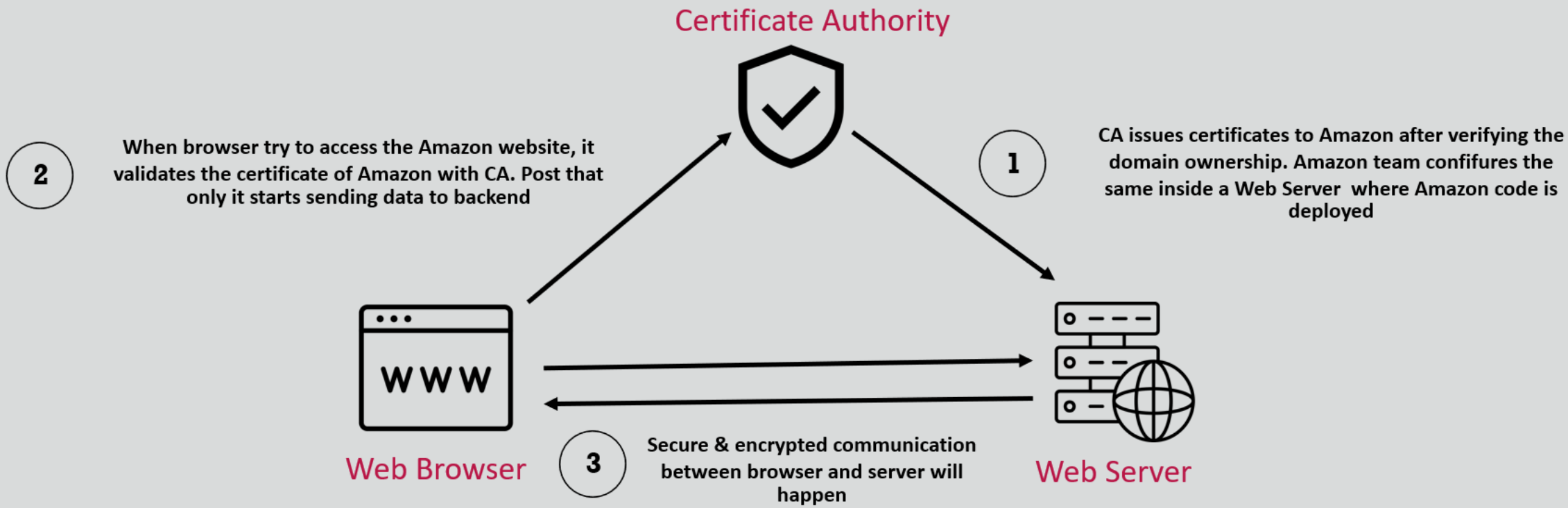
TLS is the direct successor to SSL, and all versions of SSL are now deprecated. However, it's common to find the term SSL describing a TLS connection. In most cases, the terms SSL and SSL/TLS both refer to the TLS protocol and TLS certificates.

How does TLS works ?

eazy bytes

When web browsers aim to establish a secure connection with a web server, such as https://www.amazon.com, they employ the Transport Layer Security (TLS) protocol. This not only encrypts and safeguards private communications but also validates the server's authenticity, ensuring it truly belongs to Amazon.

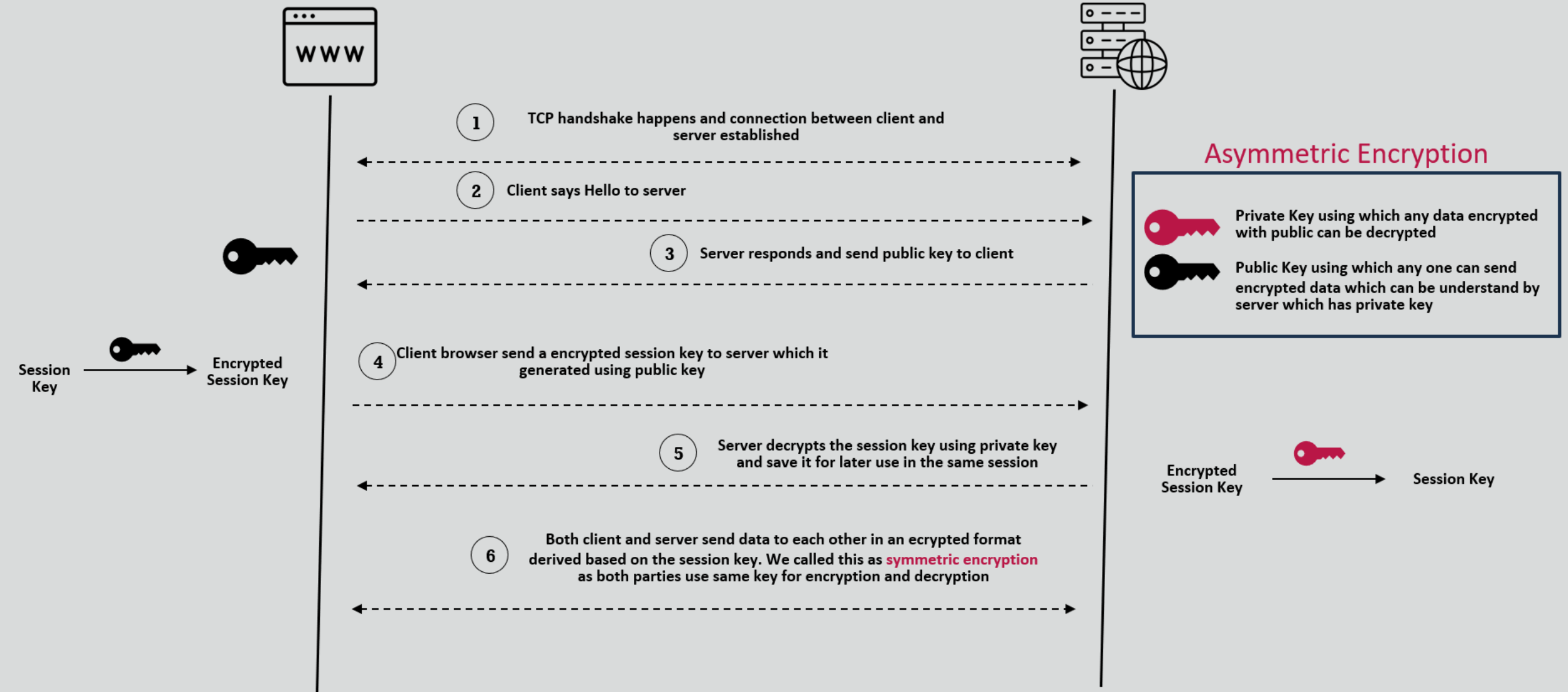
Even if we've never visited www.amazon.com before, our web browsers inherently trust the site's identity right from our initial visit. This trust is enabled by the involvement of trusted third parties (TTPs). In the context of TLS, these TTPs are known as certificate authorities (CAs), which generate and issue X.509 digital certificates to website owners after they provide proof of domain ownership.



How does TLS works ?

eazy bytes

Below are the steps happens behind the scenes just before browser start sending the data to backend server,



How Is mTLS Different from TLS ?

eazy bytes

mTLS enhances the security offered by TLS by introducing mutual authentication between the client and the server. Within the framework of mTLS, both the client and the server exchange their respective certificates and mutually confirm each other's identities prior to establishing a secure connection.

In contrast, TLS (as well as its predecessor, SSL) exclusively offers server authentication to the client. This level of authentication suffices for numerous scenarios where the client places trust in the server and aims to validate the server's identity before transmitting sensitive data.

In a zero trust security strategy, use mTLS for secure communication between controllable application components, such as microservices in a cluster. One-way TLS is typical for internet clients connecting to web services, focusing on server identification. However, with supporting tech like a service mesh, mTLS operates outside the app and simplifies implementation.

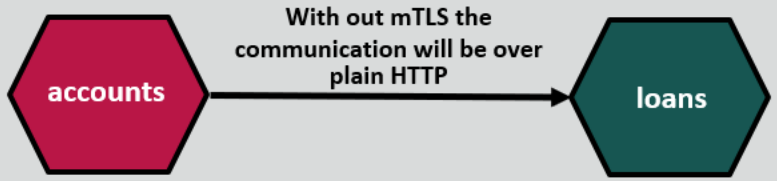
Yet, managing many certificates in mTLS can become challenging, which is where automatic mTLS through a service mesh helps ease certificate management complexity.

The organization implementing mTLS acts as its own certificate authority. This contrasts with standard TLS, in which the certificate authority is an external organization that checks if the certificate owner legitimately owns the associated domain

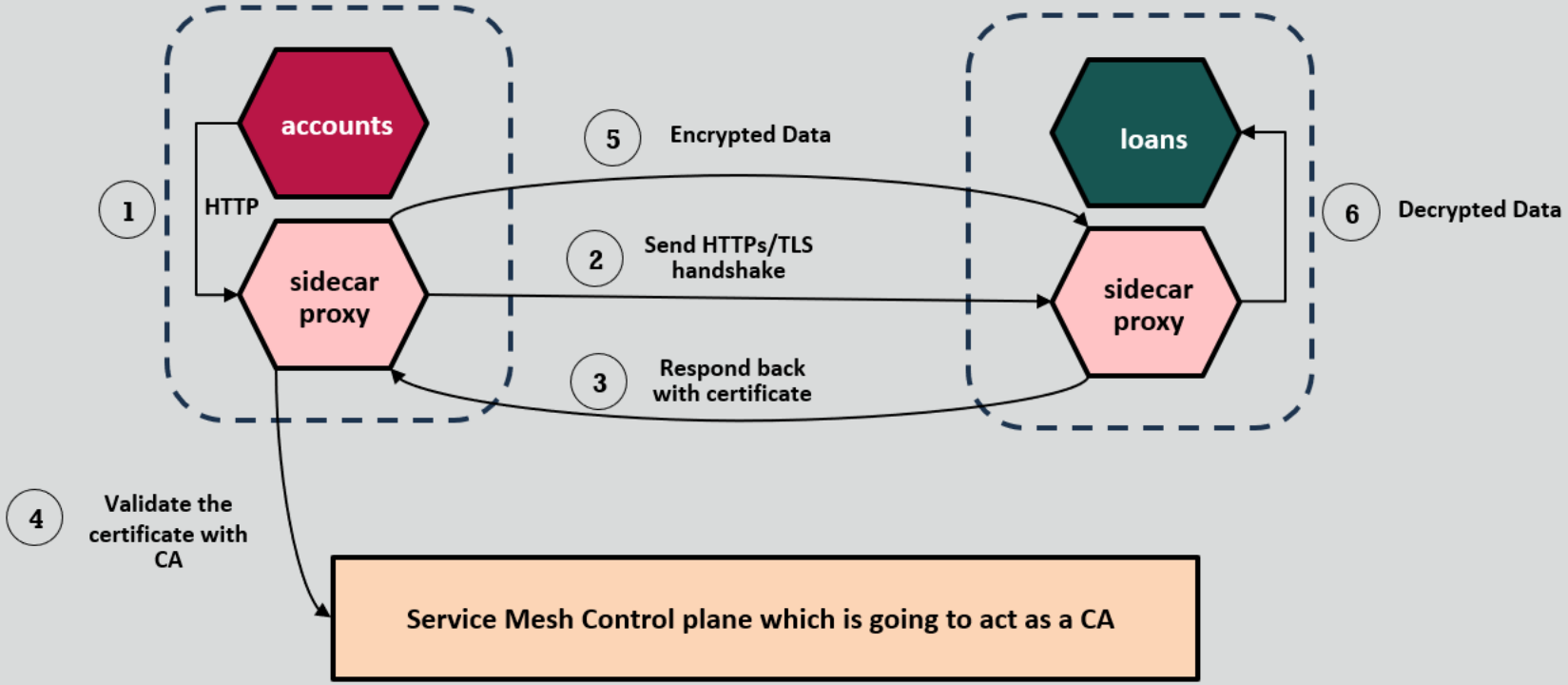
How does mTLS works ?

eazy bytes

Think like, accounts microservice container want to communicate with loans microservice container and both of them are deployed in a same Kubernetes Cluster



Sample flow of mTLS between two microservices



Why use mTLS?

eazy bytes

Mutual Transport Layer Security (mTLS) offers several advantages in securing communication between parties:

Mutual Authentication: Both the client and server authenticate each other, enhancing security by ensuring that both parties are who they claim to be. This is especially important in scenarios where both sides need to establish trust.

Protection Against Impersonation: It guards against man-in-the-middle attacks and impersonation attempts, as both parties present valid digital certificates, making it difficult for malicious actors to intercept or manipulate data.

Granular Access Control: mTLS can be used to enforce fine-grained access control, allowing organizations to specify which clients are permitted to access specific services or resources, based on the certificates they possess.

Resistance to Credential Compromise: Unlike username/password authentication, mTLS relies on cryptographic keys stored securely, making it less susceptible to credential theft or brute-force attacks.

Simplified Key Management: In many cases, mTLS leverages digital certificates issued by trusted certificate authorities (CAs), reducing the complexity of managing encryption keys and ensuring their validity.

Scalability: While managing certificates can be challenging at scale, tools like service meshes can automate certificate distribution and rotation, simplifying operations in large deployments.

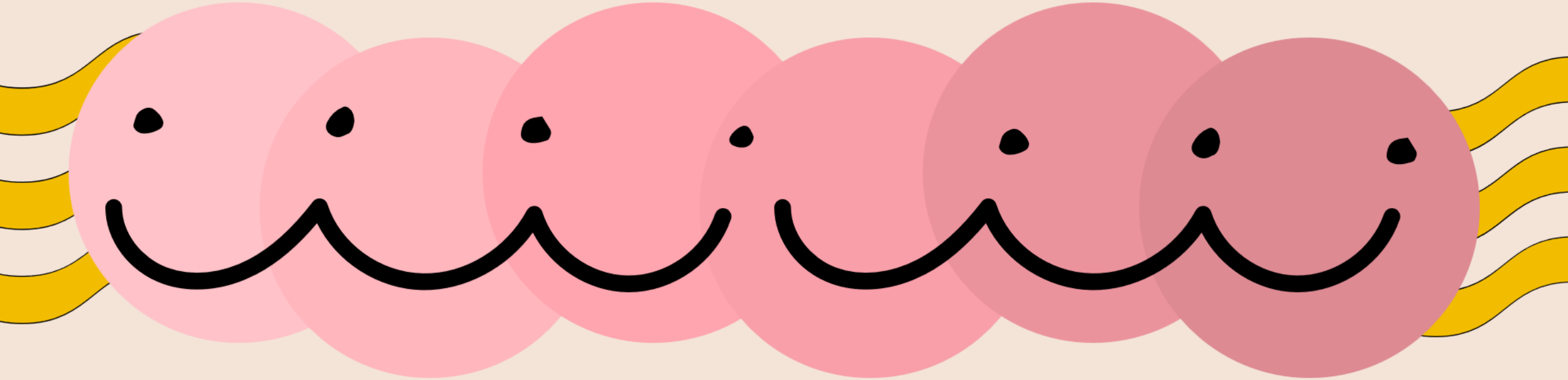
Compliance: mTLS helps organizations meet regulatory compliance requirements for securing sensitive data and privacy, such as those outlined in GDPR, HIPAA, or PCI DSS.

Zero Trust Security: mTLS aligns with the principles of zero trust security by requiring verification at every step of communication, fostering a more secure and less trusting network environment.

eazy bytes

CONGRATULATIONS

YOU ARE NOW A MASTER OF MICROSERVICES USING SPRING, DOCKER AND KUBERNETES



A BIG THANK YOU

HOPING FOR OUR PATHS TO CROSS AGAIN