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

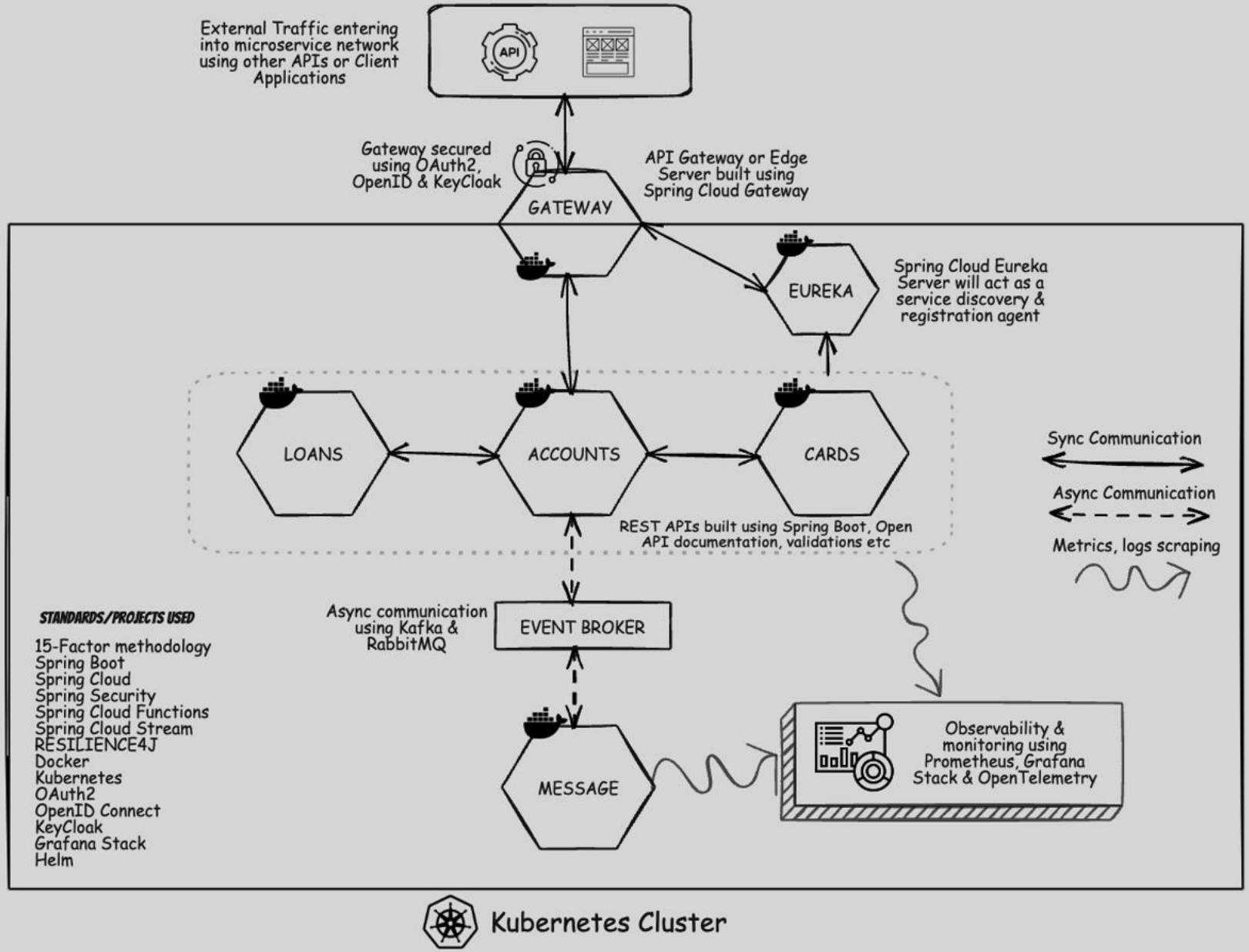
Deep dive on Helm (kubernetes package manager)

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



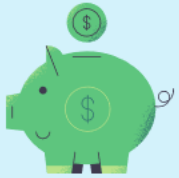
What are Microservices ?

eazy bytes

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

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

Cards

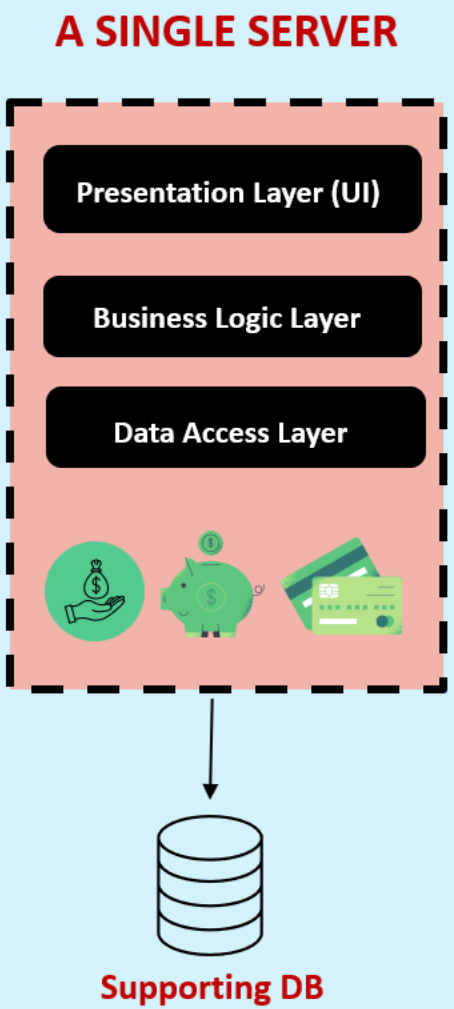


Accounts

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

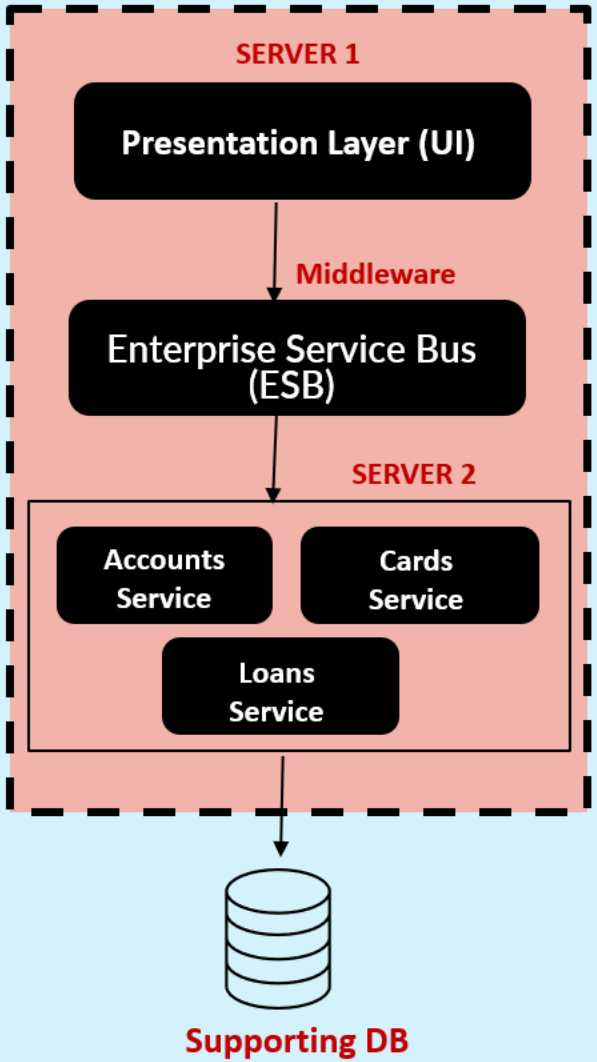
Spring Data JPA/ORM

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

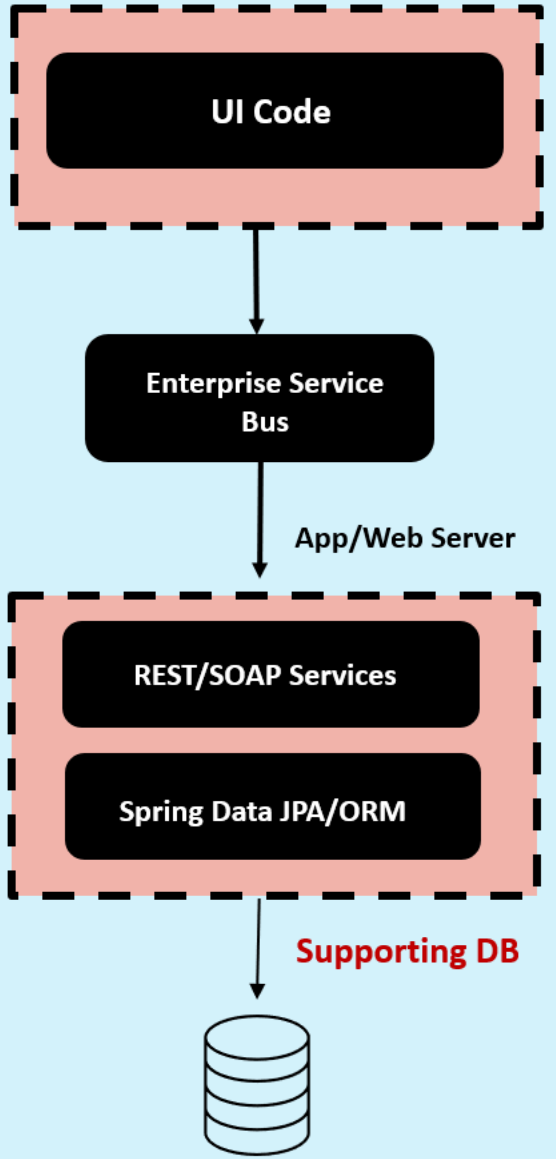
Accounts Dev Team

Loans Dev Team

Cards Dev Team

UI Code repo

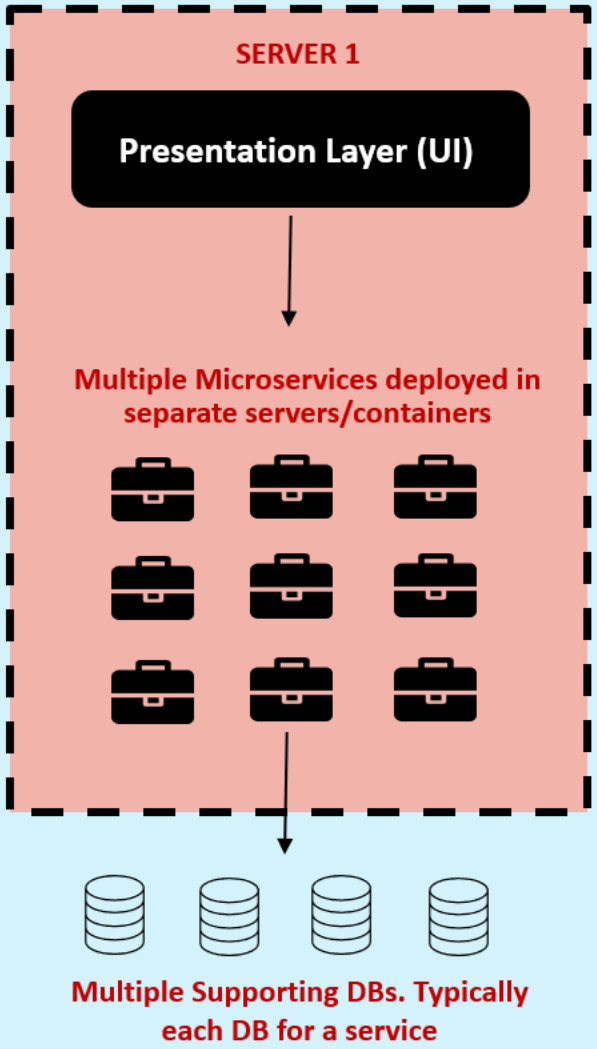
Backend Services repo



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

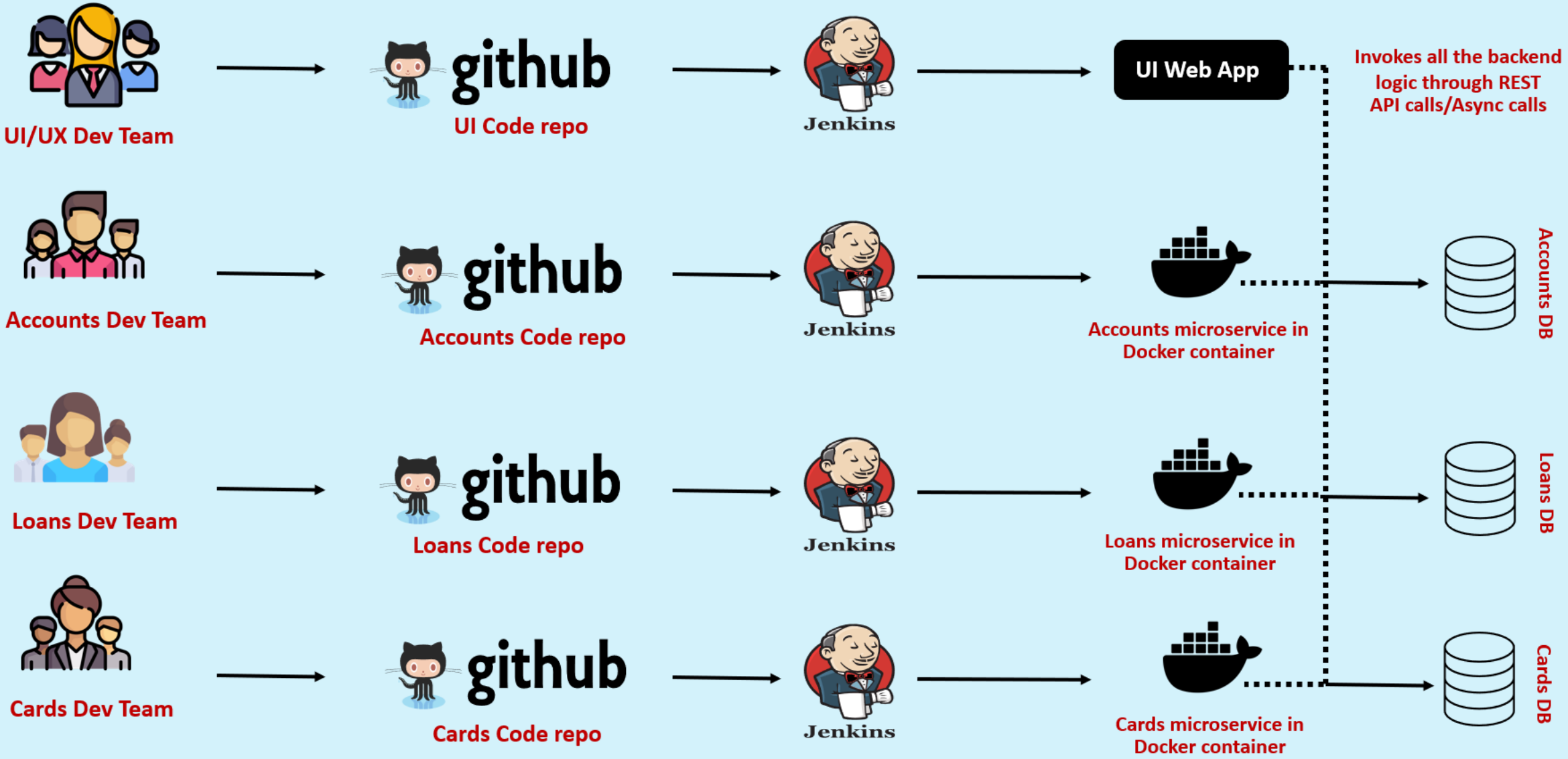
Cons

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.

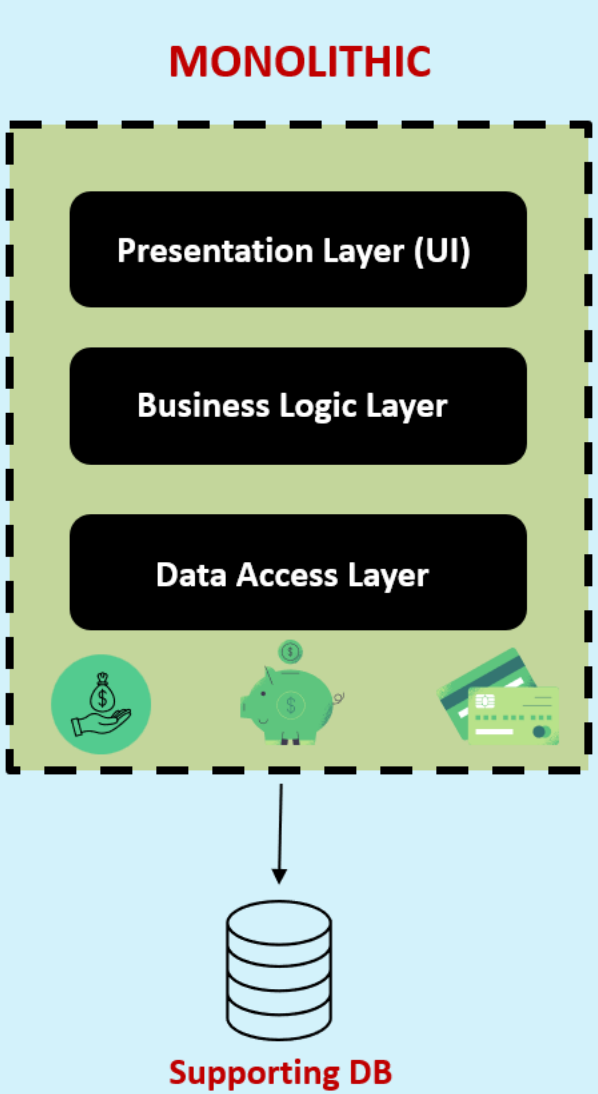
The GREAT MICROSERVICES

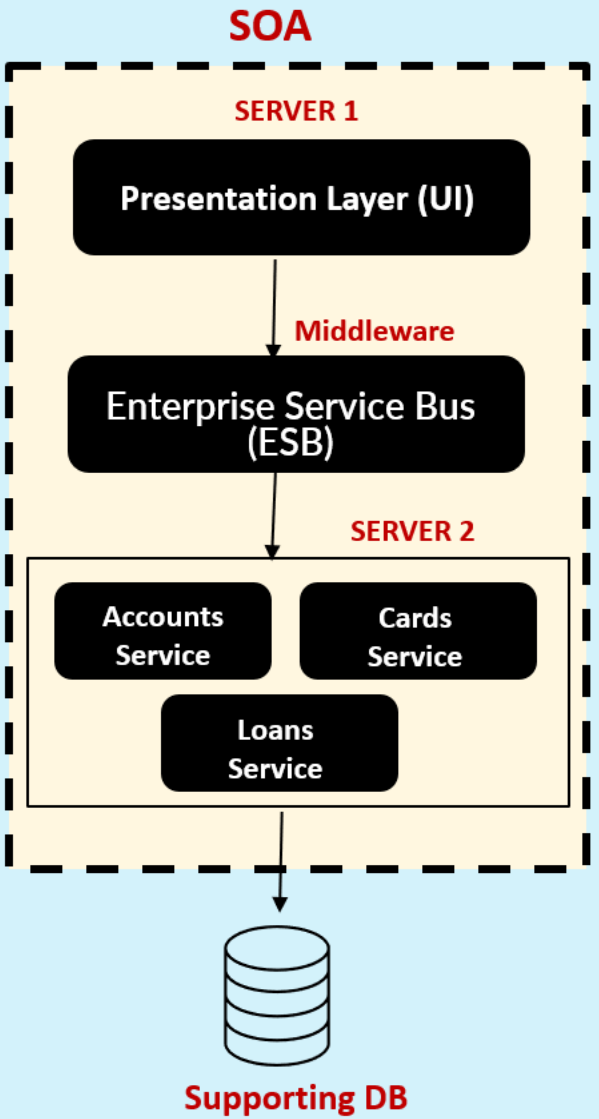
eazy bytes

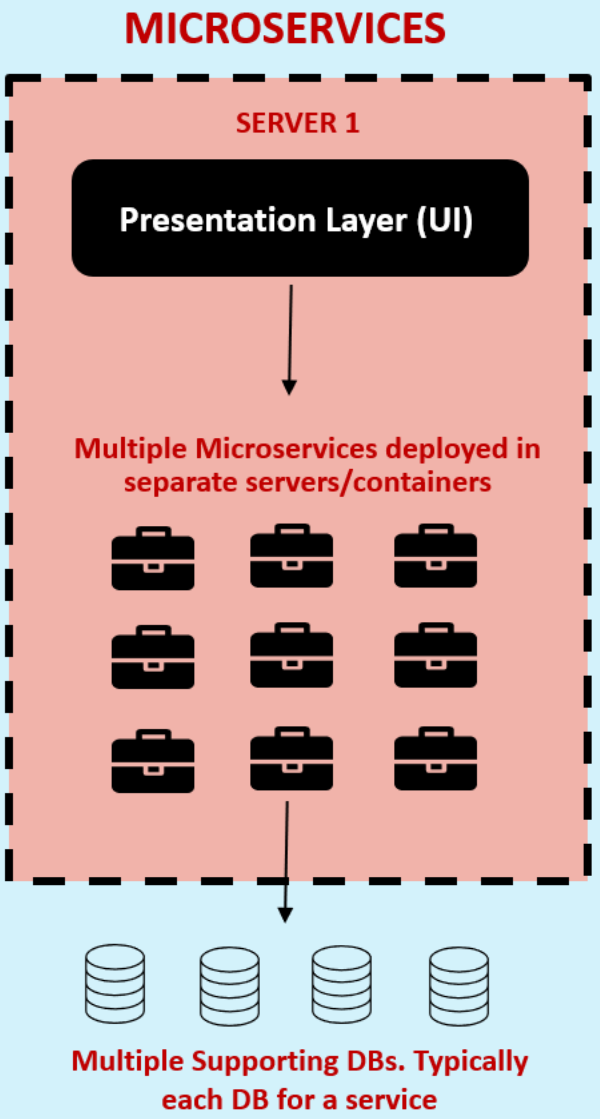


MONOLITHIC vs SOA vs MICROSERVICES

eazy bytes

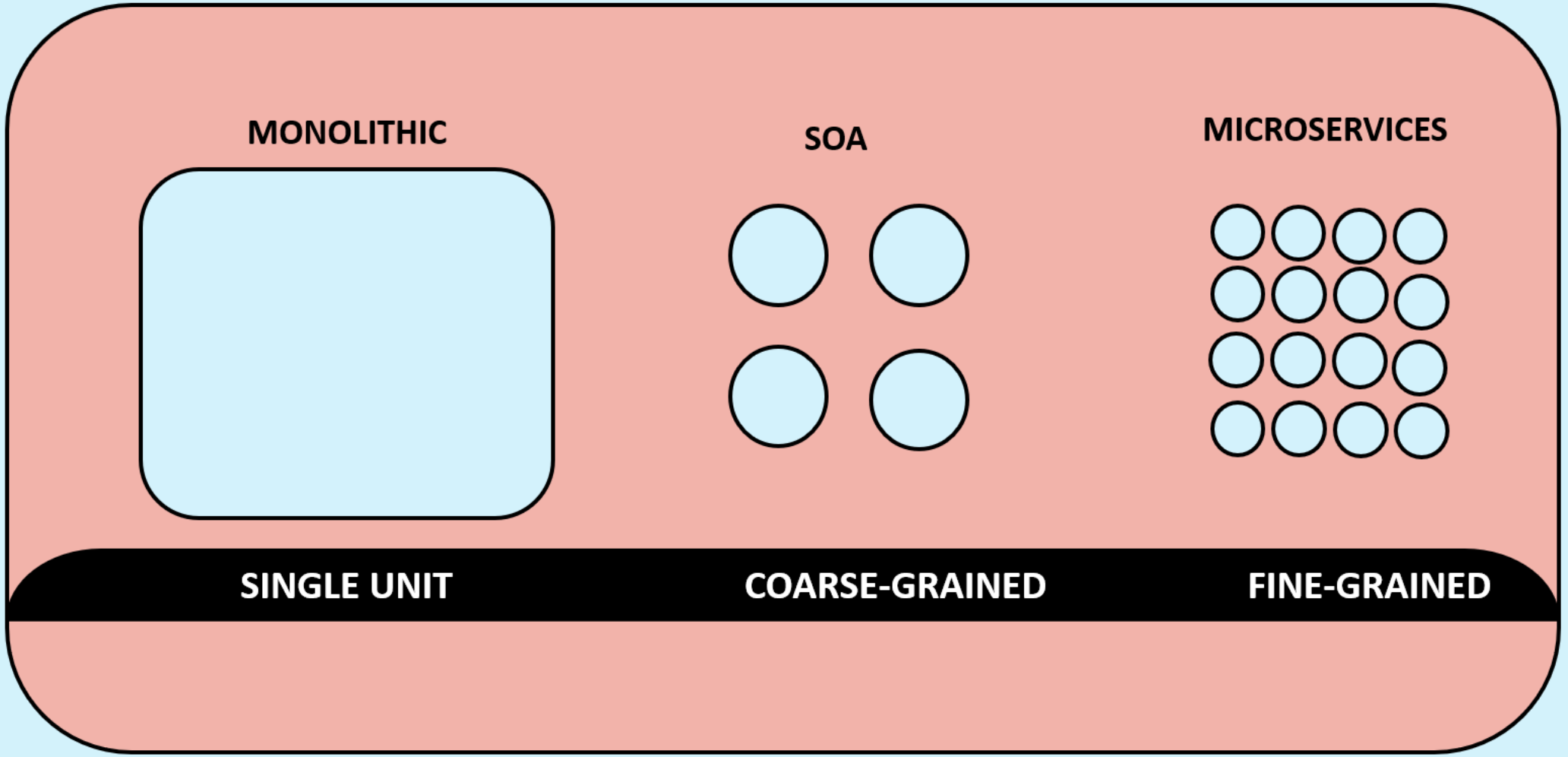






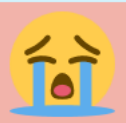
MONOLITHIC vs SOA vs MICROSERVICES

eazy bytes



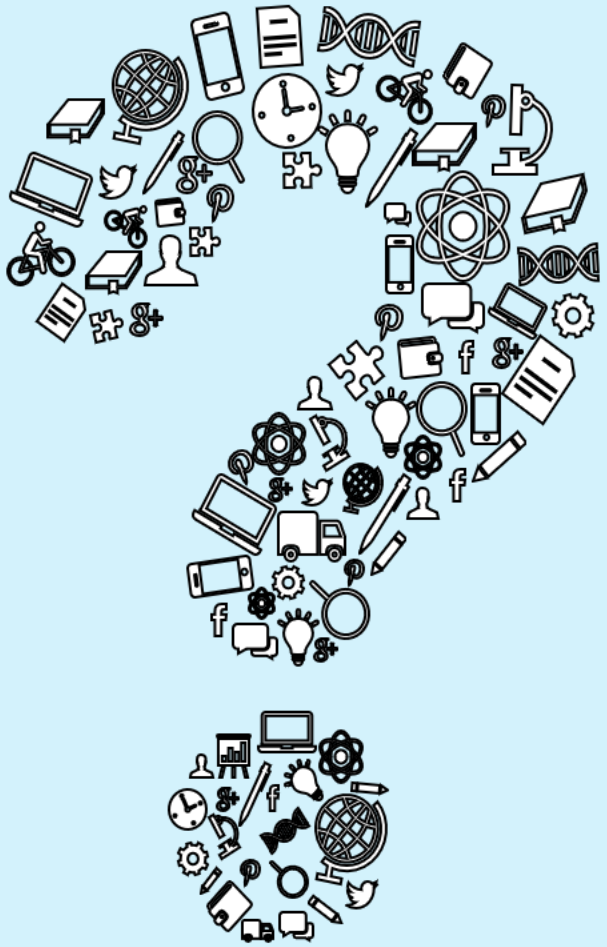
MONOLITHIC VS SOA VS MICROSERVICES

eazy bytes

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

DEFINITION OF MICROSERVICE ?

eazy bytes

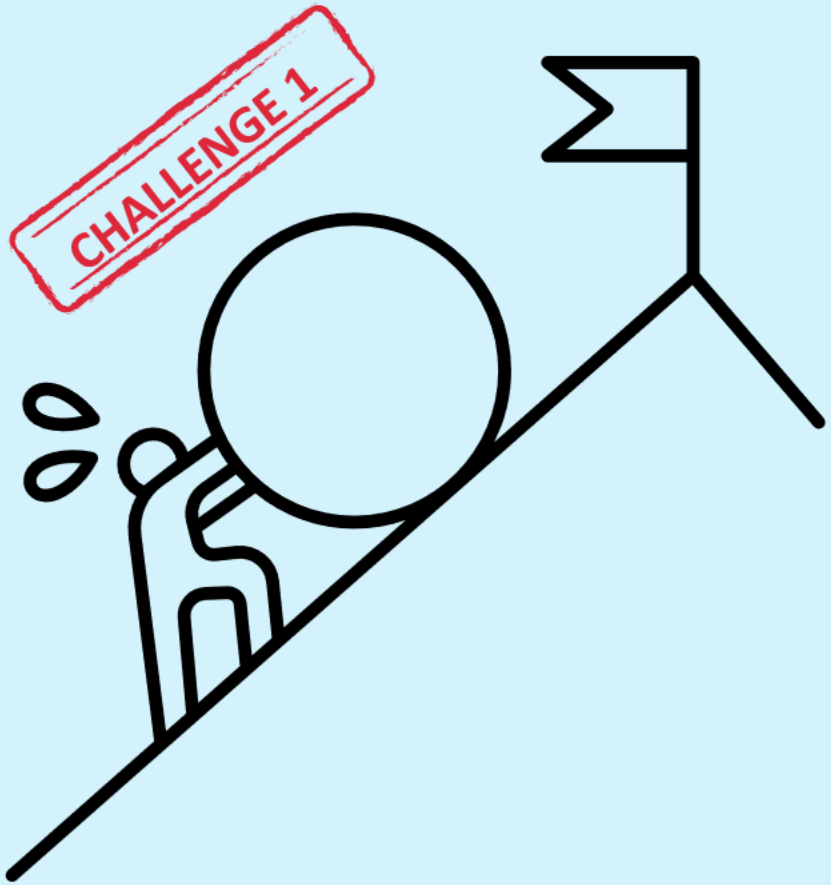


“Microservices is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, built around business capabilities and independently deployable by fully automated deployment machinery.”

- 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

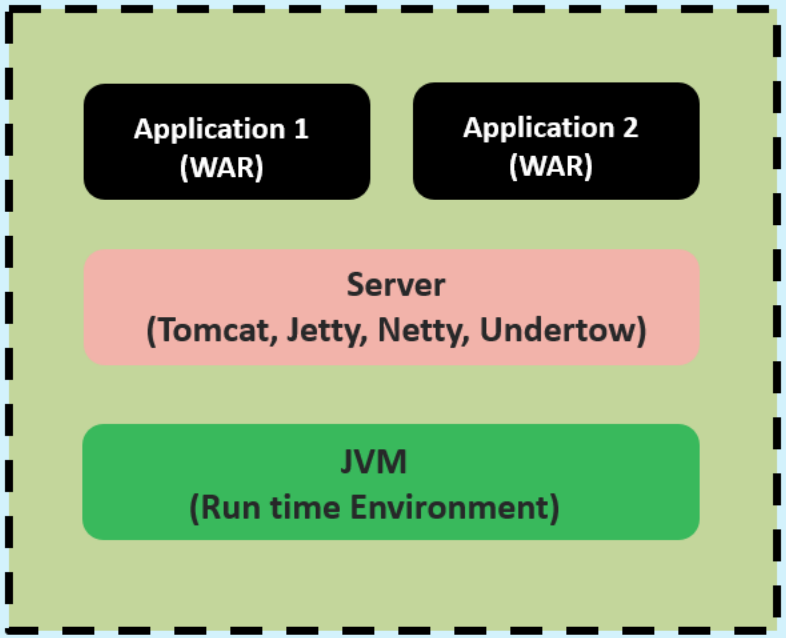
Well-suited for cloud-native development. It integrates smoothly with cloud platforms like Kubernetes, provides support for containerization, and enables seamless deployment to popular cloud providers

WHY SPRING BOOT FOR MICROSERVICES?

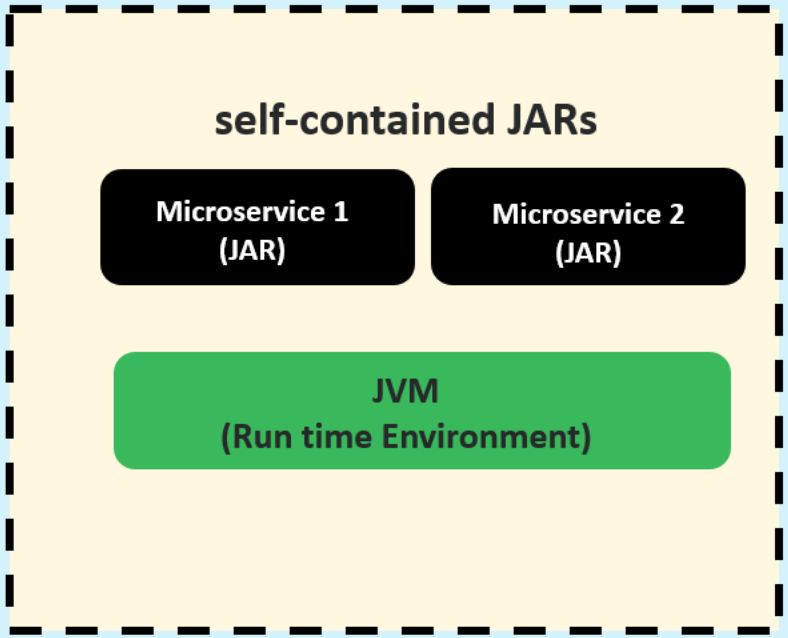
eazy bytes

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)



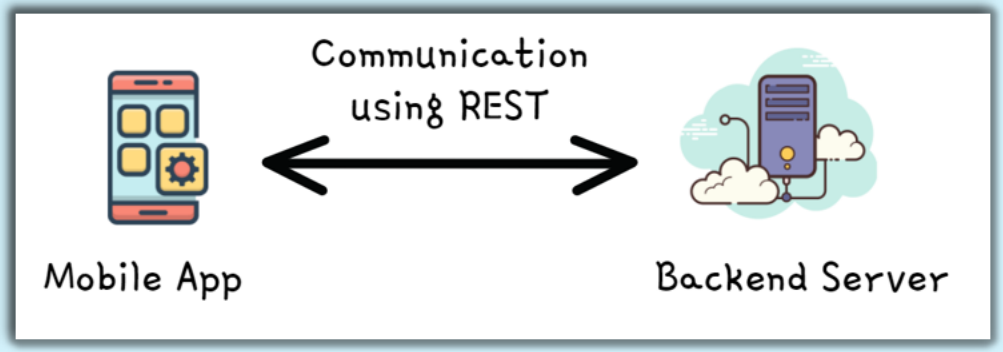
JARs with embedded server
(Modern approach)

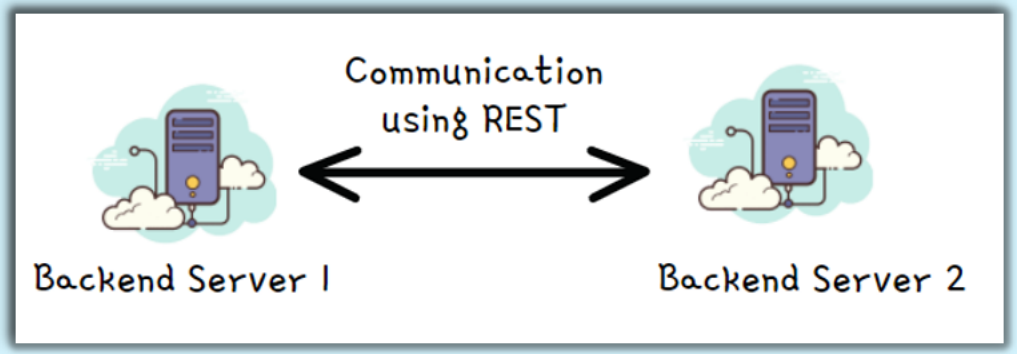
Implementing REST Services

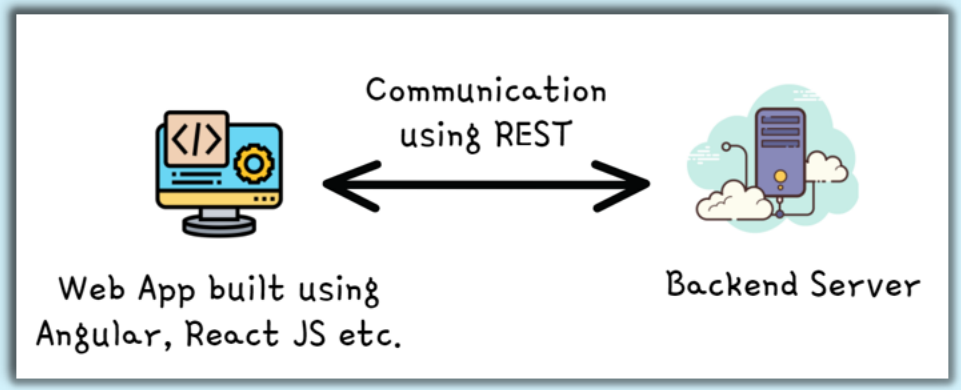
eazy bytes

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

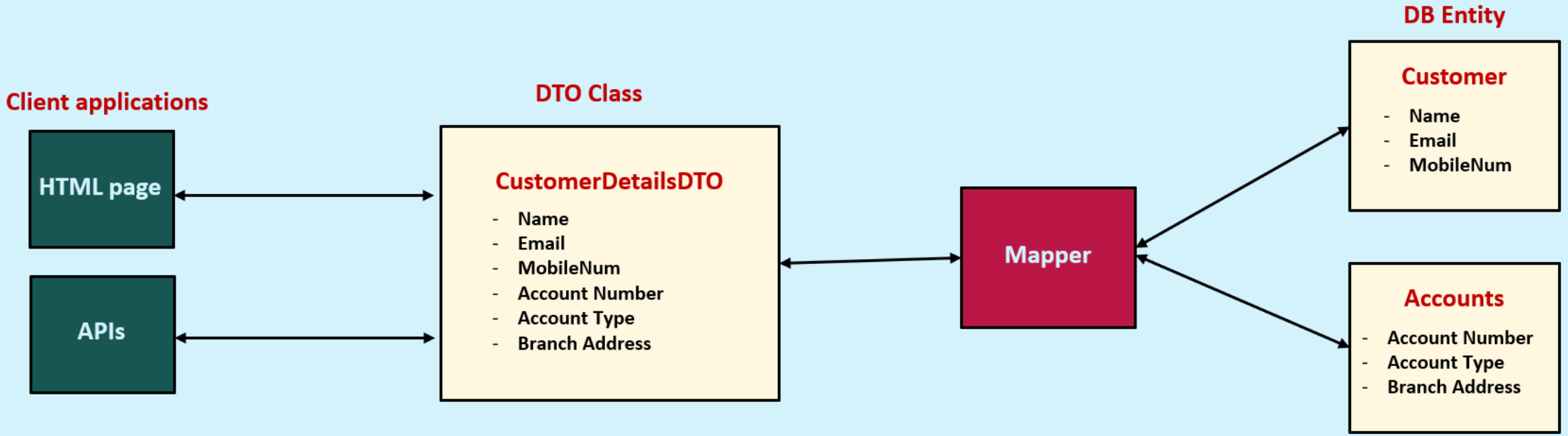
Document REST Services

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.

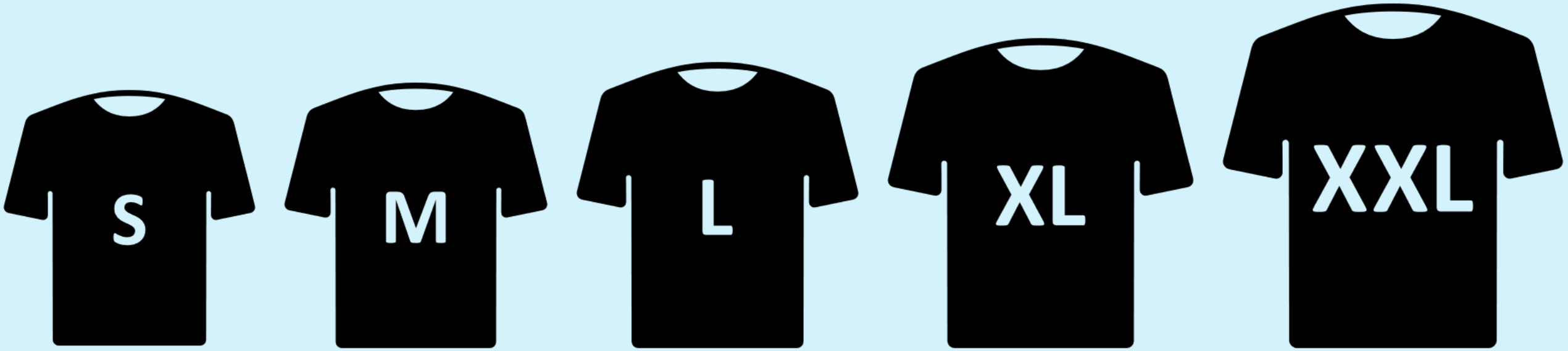
06 Perform auditing using Spring Data JPA

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

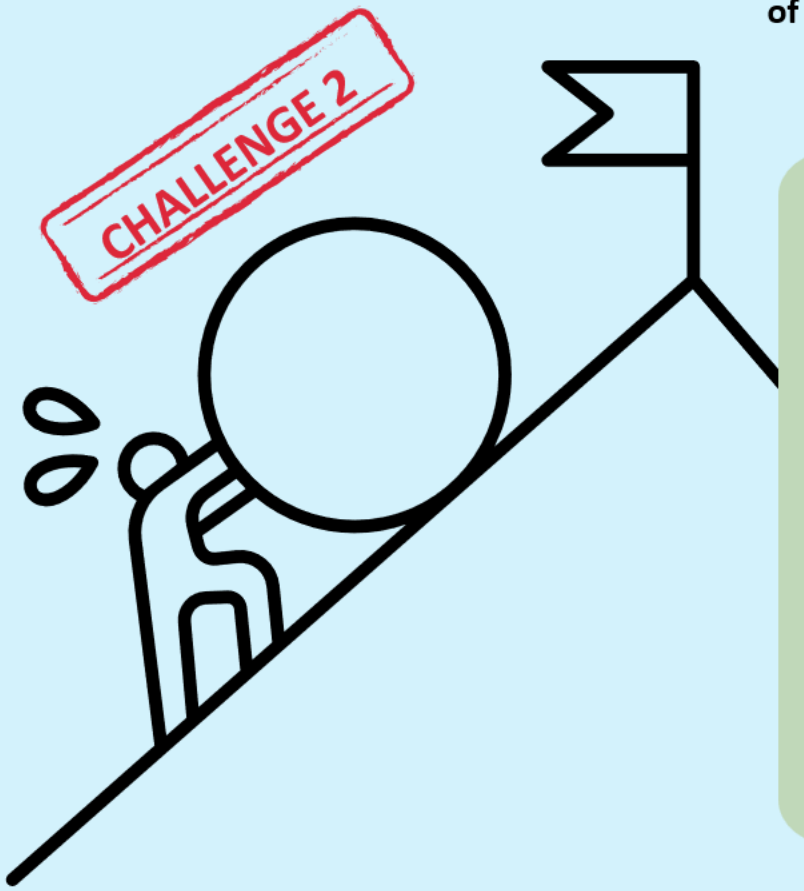
With the help of OpenAPI specifications, Swagger, Spring Doc library, we documented our REST APIs. In the same process, we used annotations like **@Schema, @Tag, @Operation, @ApiResponse** etc.

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

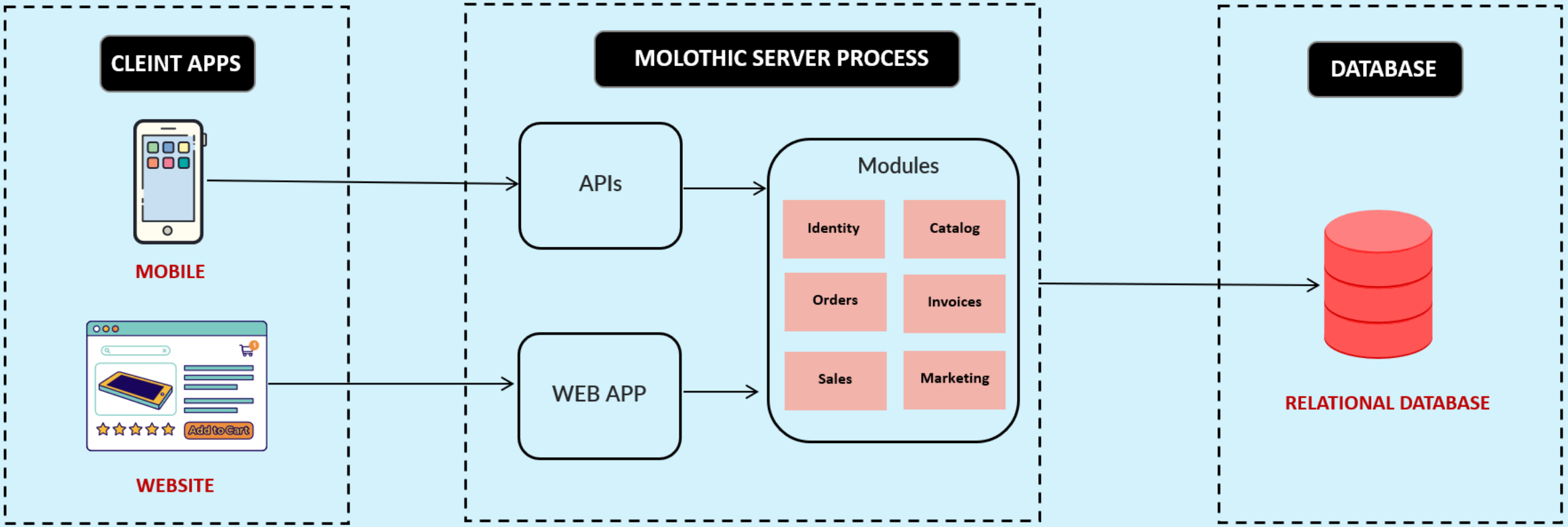
NOT CORRECT SIZING AS WE CAN SEE TOO MANY SERVICES UNDER LOANS & CARDS

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

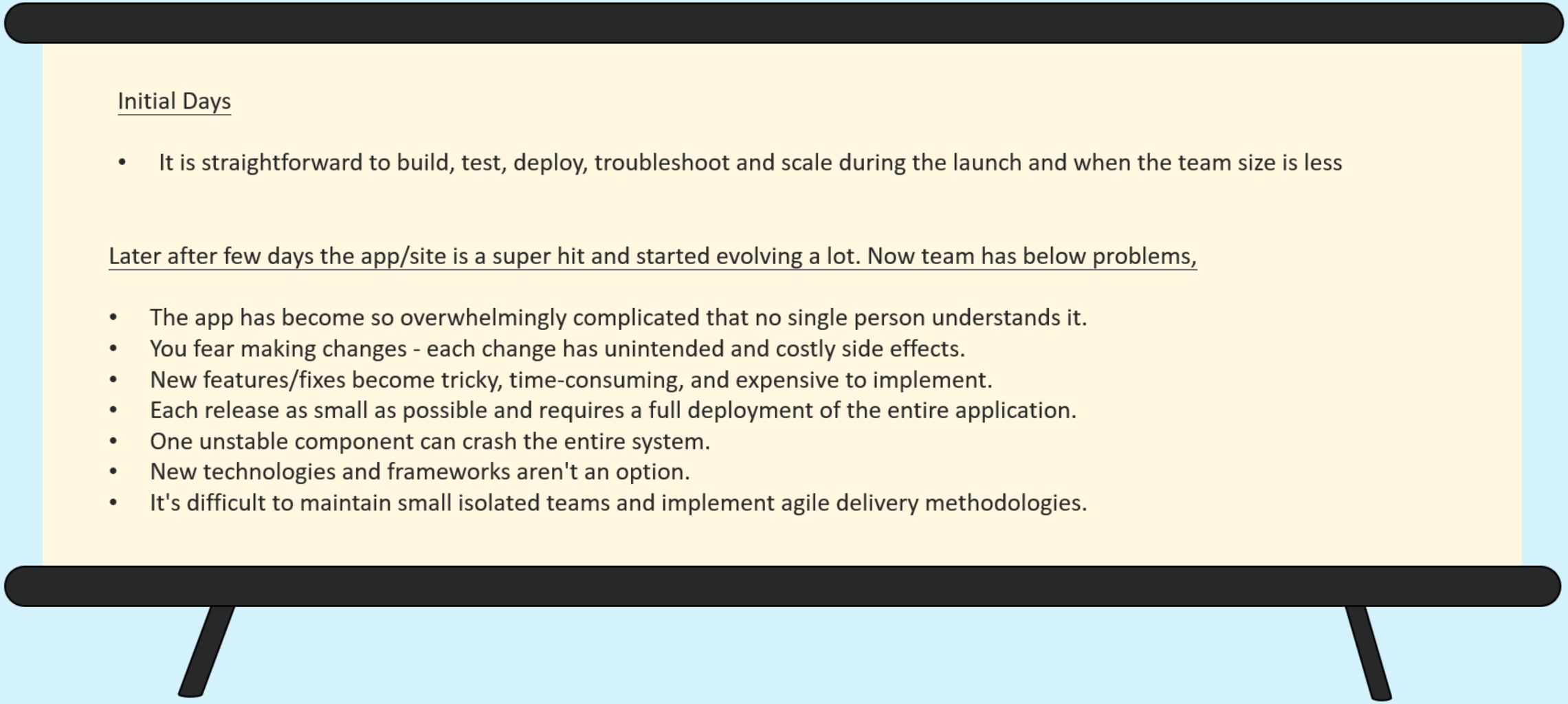


MONOLOTHIC TO MICROSERVICES

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,

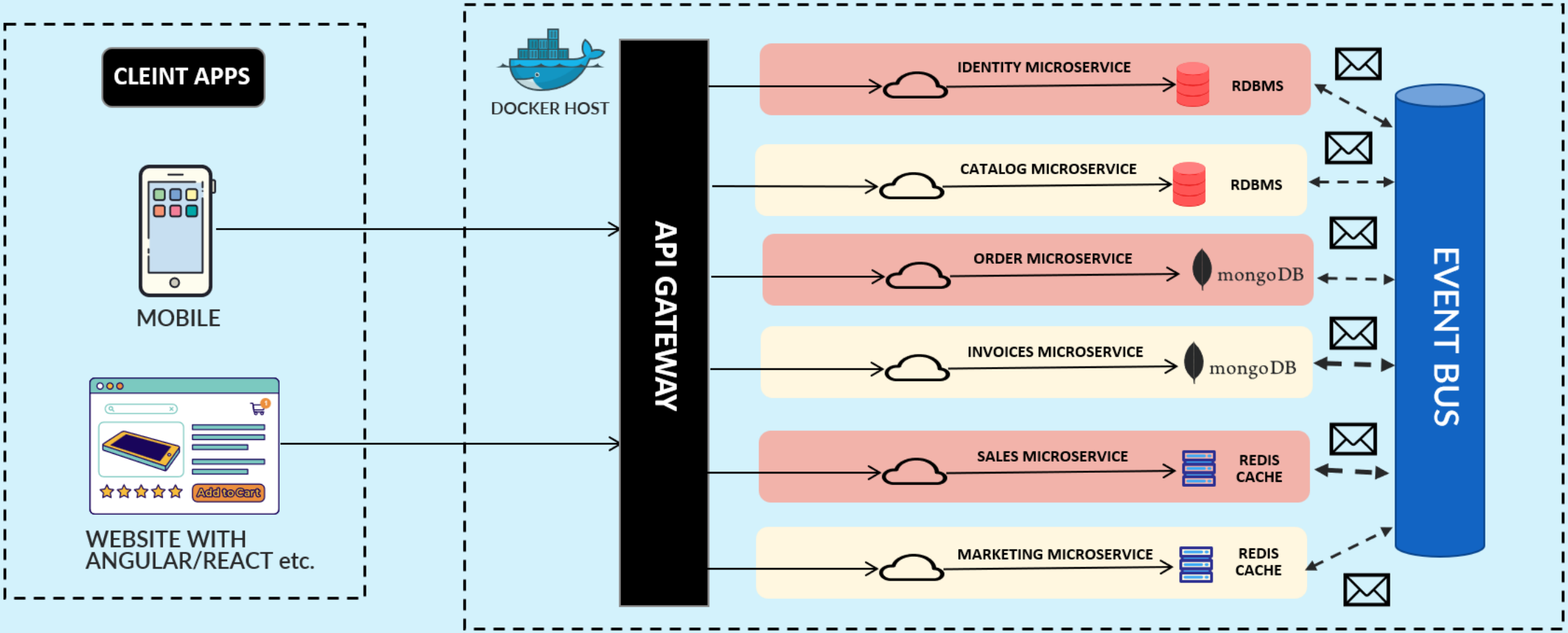
- The app has become so overwhelmingly complicated that no single person understands it.
- You fear making changes - each change has unintended and costly side effects.
- New features/fixes become tricky, time-consuming, and expensive to implement.
- Each release as small as possible and requires a full 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.



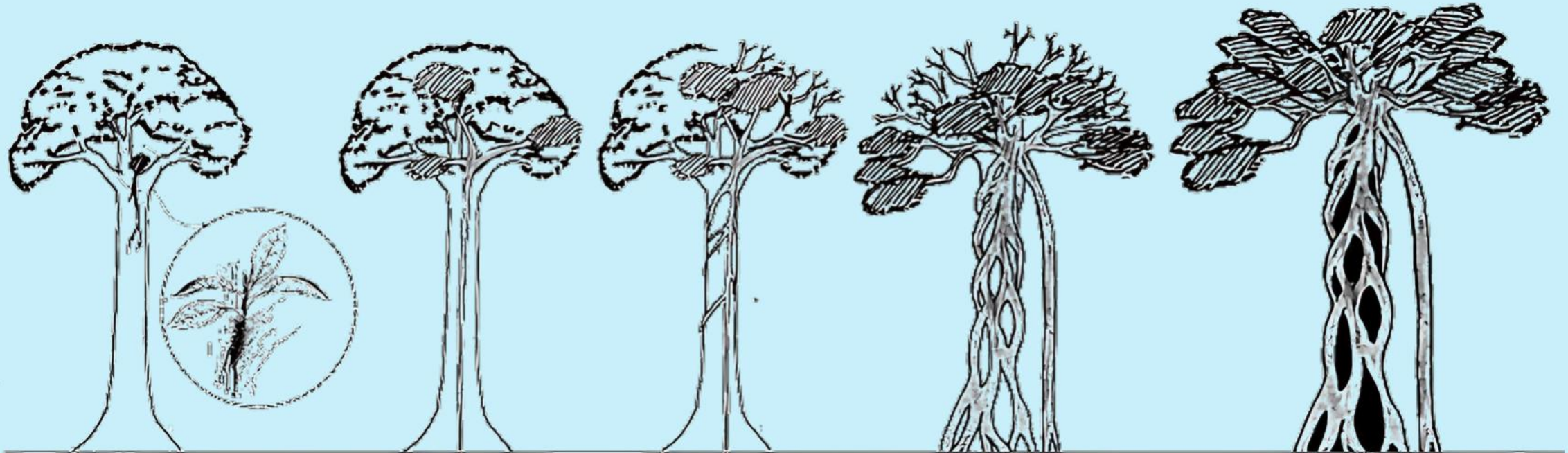
Strangler Fig pattern

eazy bytes

The Strangler Fig Pattern is a software migration pattern used to gradually replace or refactor a legacy system with a new system, piece by piece, without disrupting the existing functionality. This pattern gets its name from the way a strangler fig plant grows around an existing tree, slowly replacing it until the original tree is no longer needed.

When to Use the Strangler Fig Pattern:

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

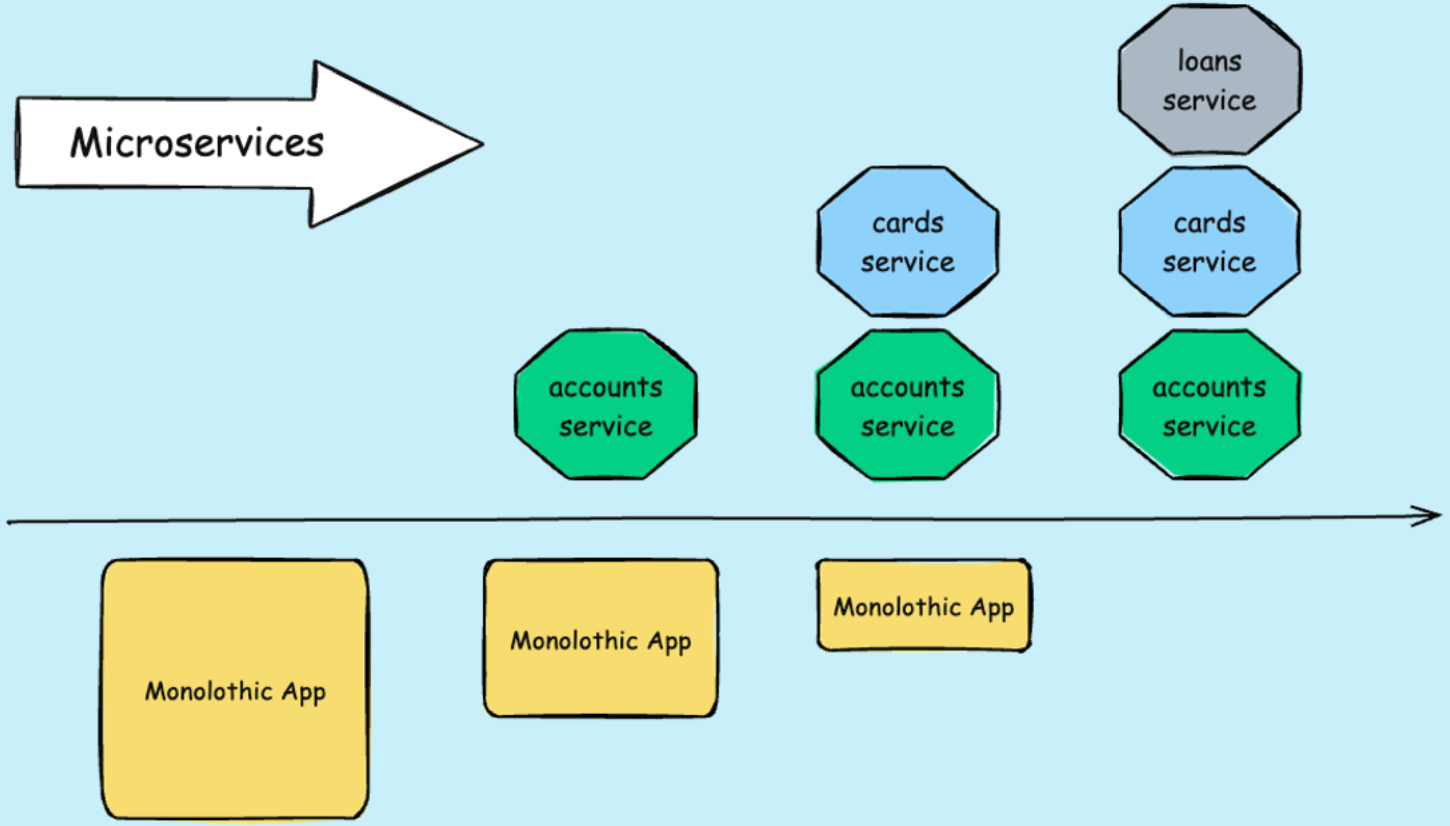


Strangler Fig pattern

eazy bytes

The Strangler Fig Pattern facilitates the migration of a monolithic application to a modern microservices architecture by leveraging a Domain-Driven Design (DDD) approach.

The legacy monolith is carefully analyzed, broken down into distinct domains, and services are gradually rewritten using newer technologies. This incremental transformation ensures that each service is refactored independently, allowing for a smooth transition from the monolith to a fully microservices-based architecture while maintaining system functionality throughout the process.

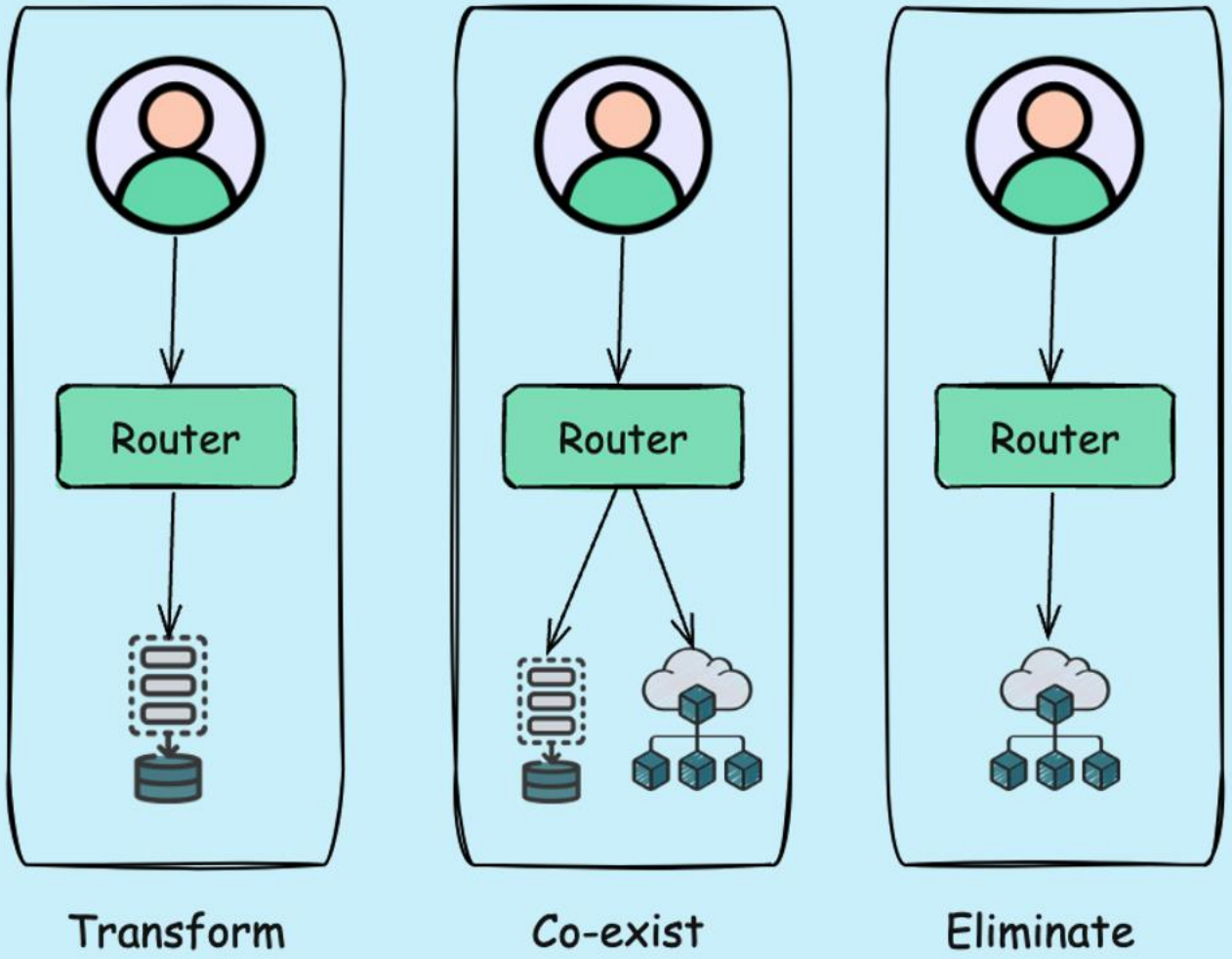


Strangler Fig pattern

eazy bytes

Next, a **Strangler Facade** is introduced to manage traffic routing between the legacy monolithic application and the new microservices. This routing can also be handled via an API Gateway, if preferred. During this phase, both the monolith and microservices coexist.

As the microservices are fully developed and stabilized, the corresponding functionality in the monolith is gradually phased out or "strangled" and eventually removed. The process involves four key steps: **identification**, **transformation**, **co-existence**, and **elimination**.



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

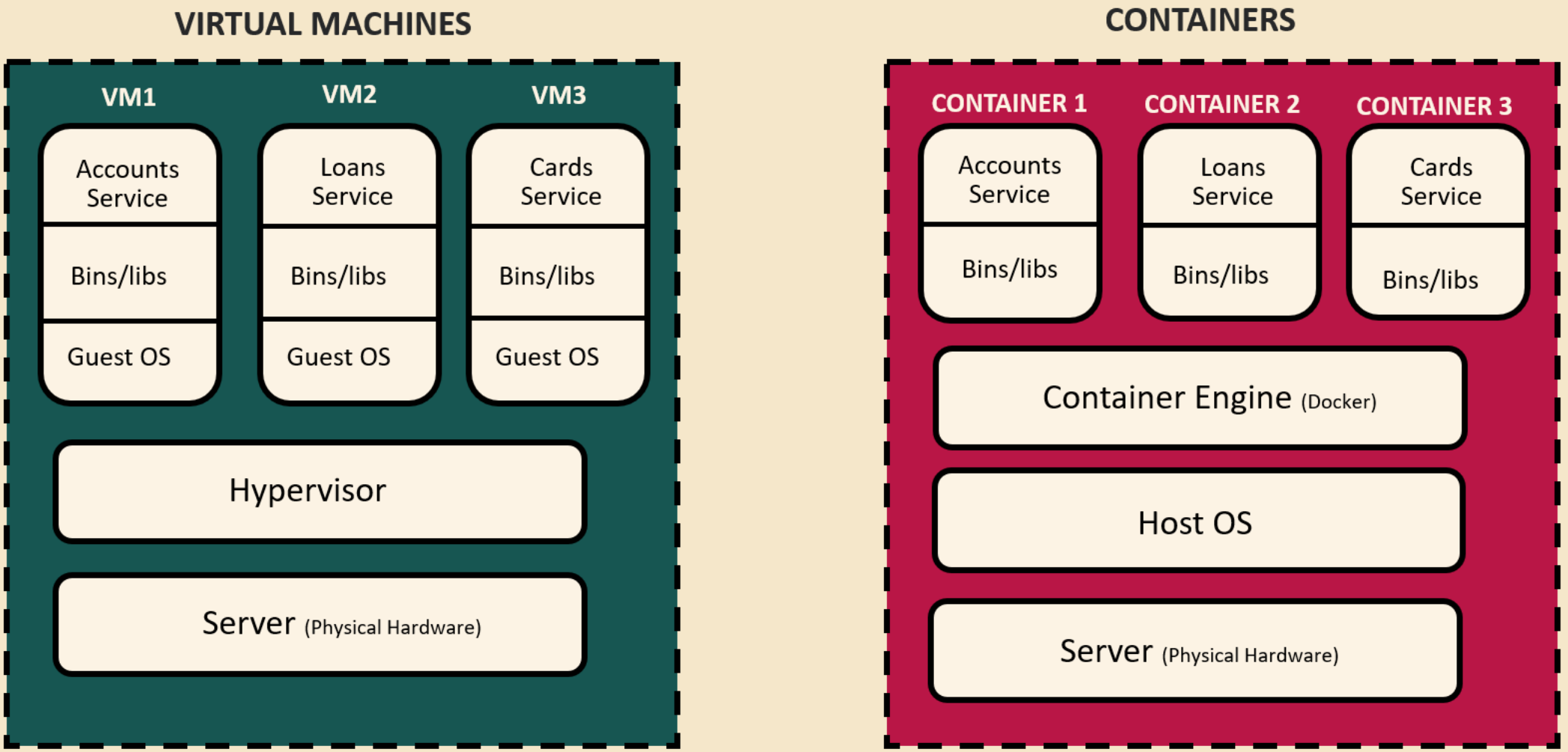
How do we scale our applications based on the demand on the fly with minimum effort & cost?

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.

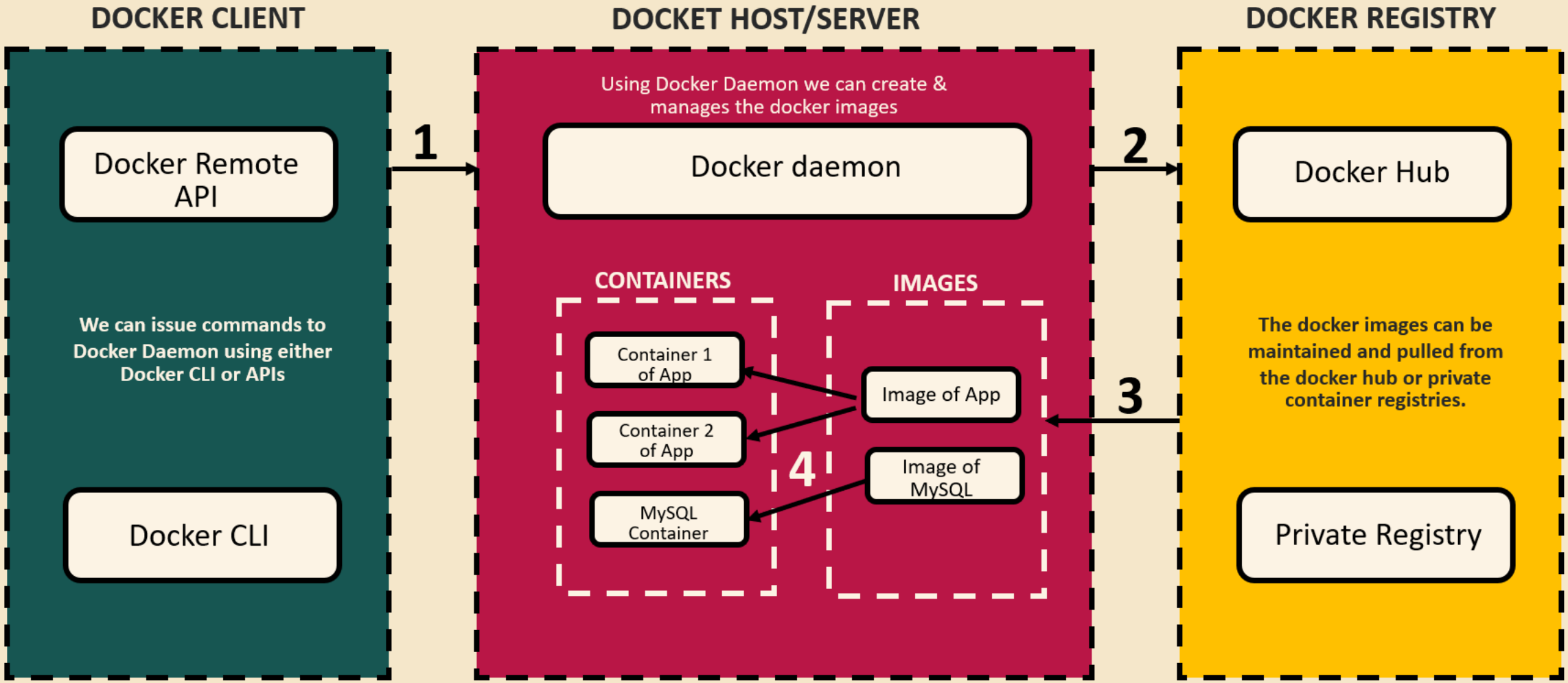
As a user, your experience will be very similar to using Docker on a Linux machine, with minimal noticeable differences. However, when you utilize the Docker CLI to execute commands, you are actually interacting with a Docker server running on a separate machine, which in this case is the Linux-based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

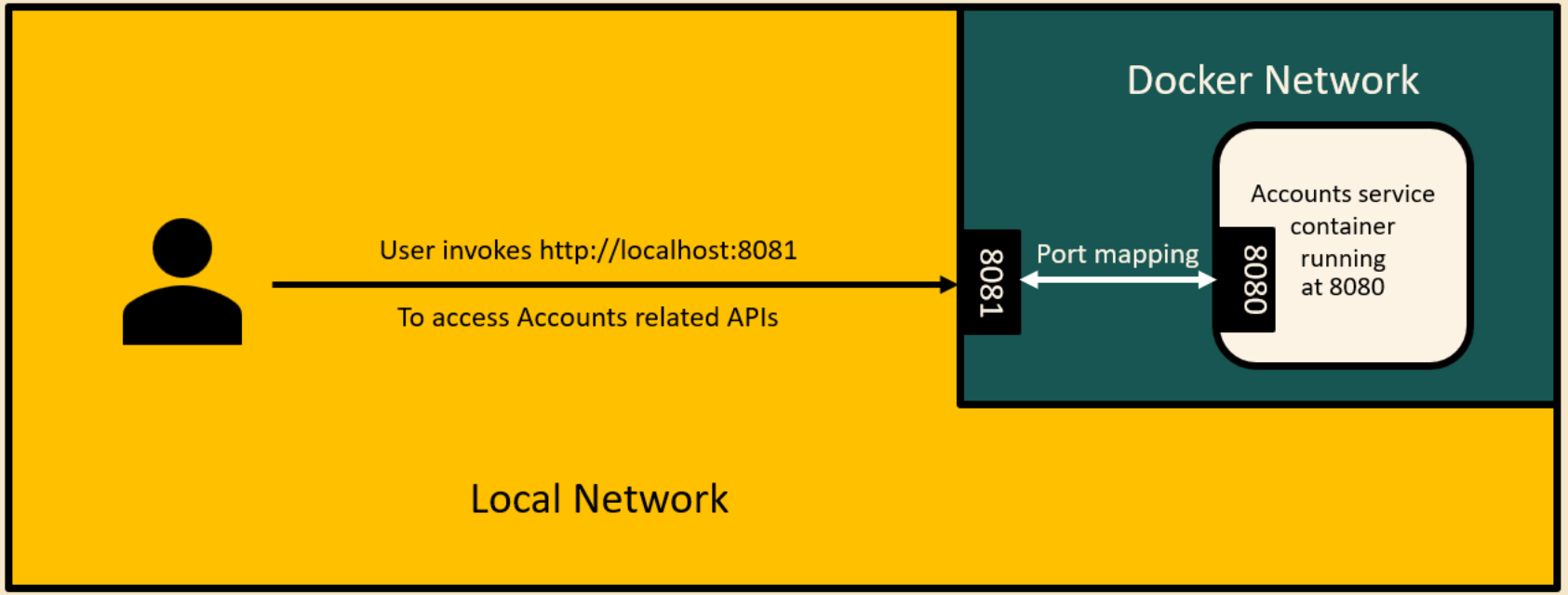
```
#Start with a base image containing Java runtime
FROM openjdk:17-jdk-slim

#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

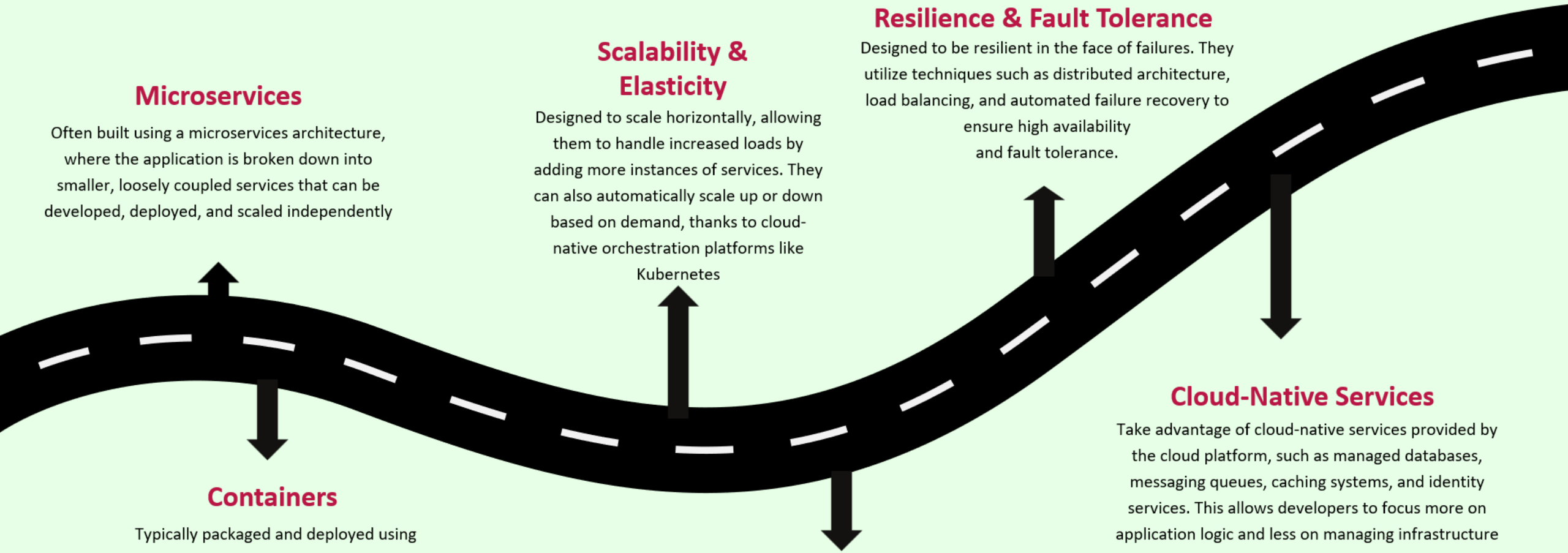
Often built using a microservices architecture, where the application is broken down into smaller, loosely coupled services that can be developed, deployed, and scaled independently

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

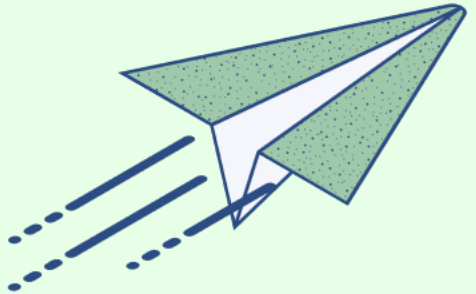
11 Port binding

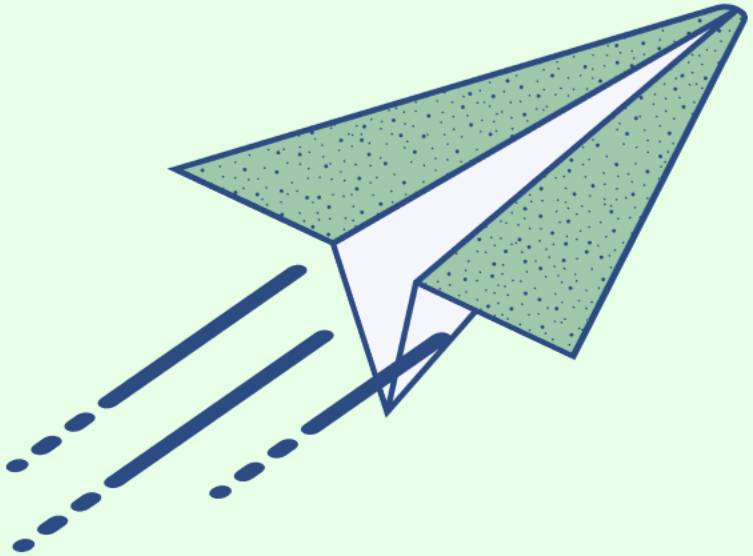
12 Stateless processes

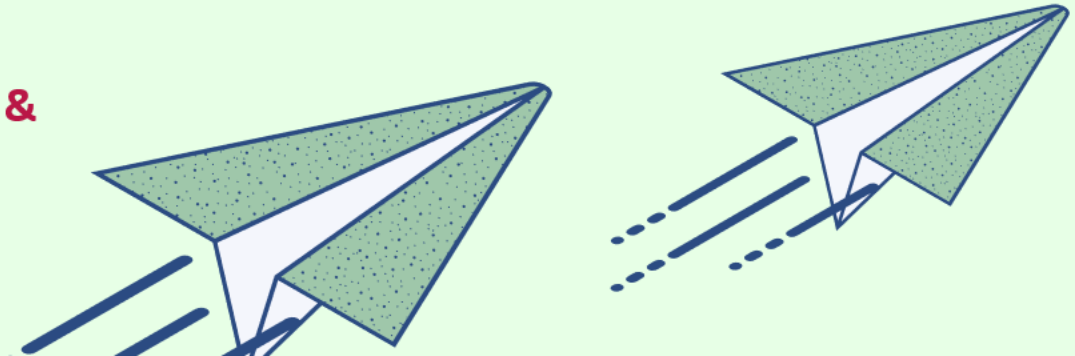
13 Concurrency

14 Telemetry

15 Authentication & authorization





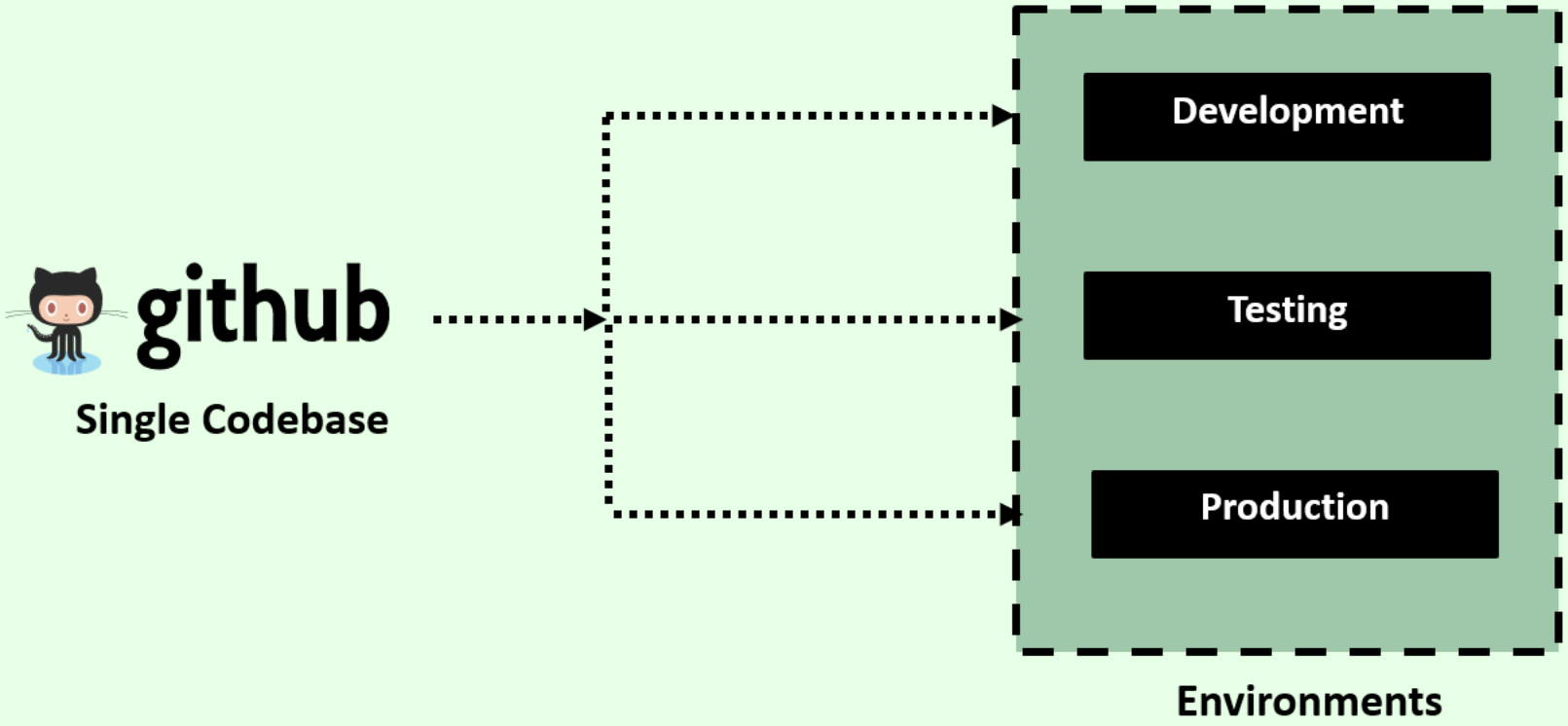


15-Factor methodology - One codebase, one application

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.

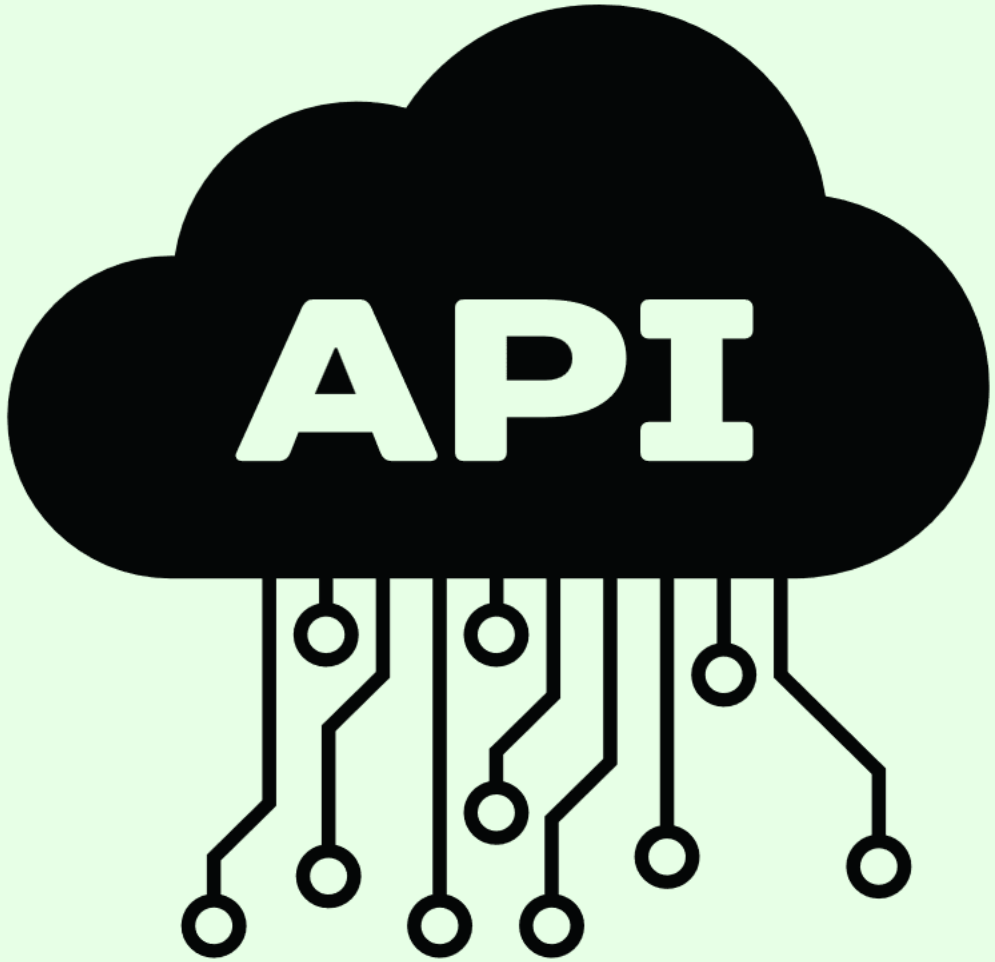


15-Factor methodology - API first

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

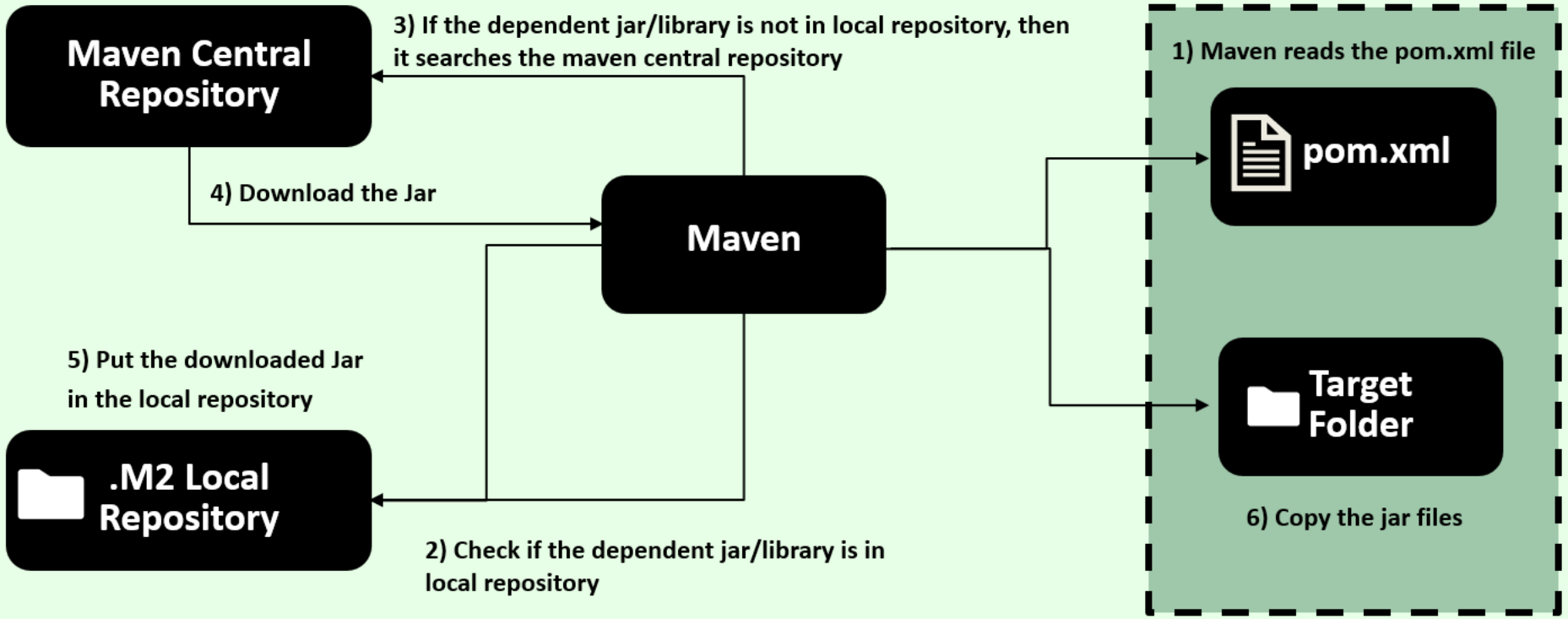
This upfront design of the API contract results in more reliable and testable integration with other systems as part of the deployment pipeline. Moreover, internal modifications to the API implementation can be made without impacting other applications or teams that rely on it



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.

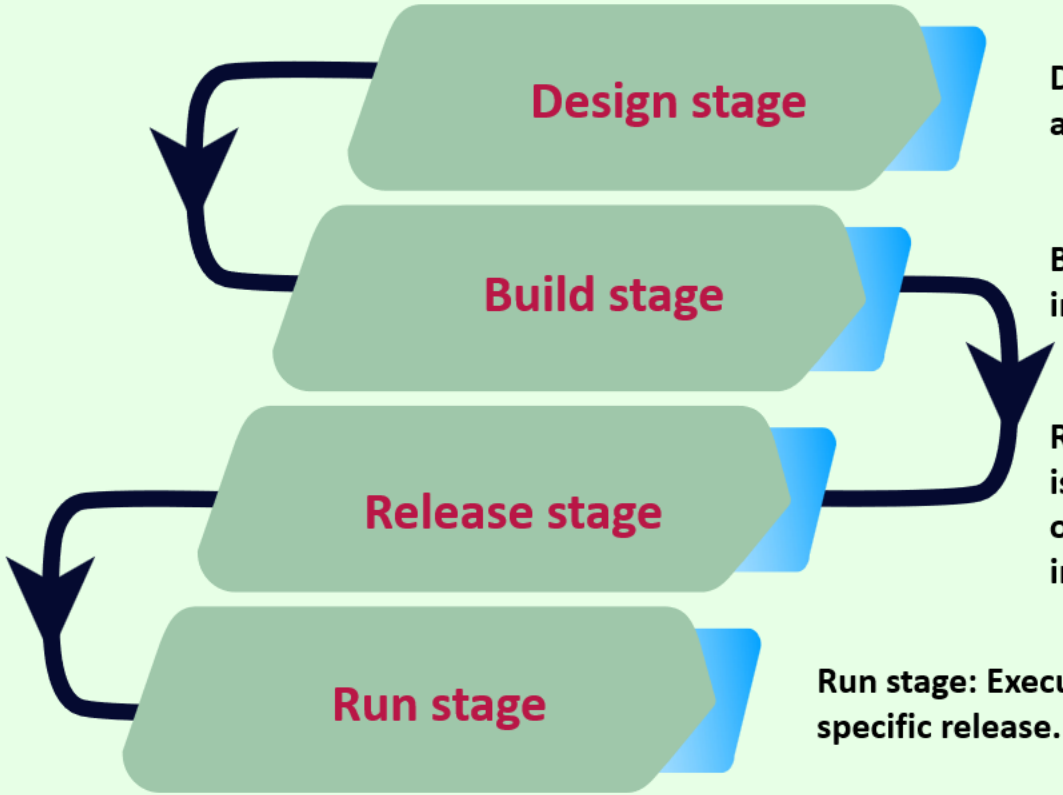
Sample flow when we use Maven as build tool



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

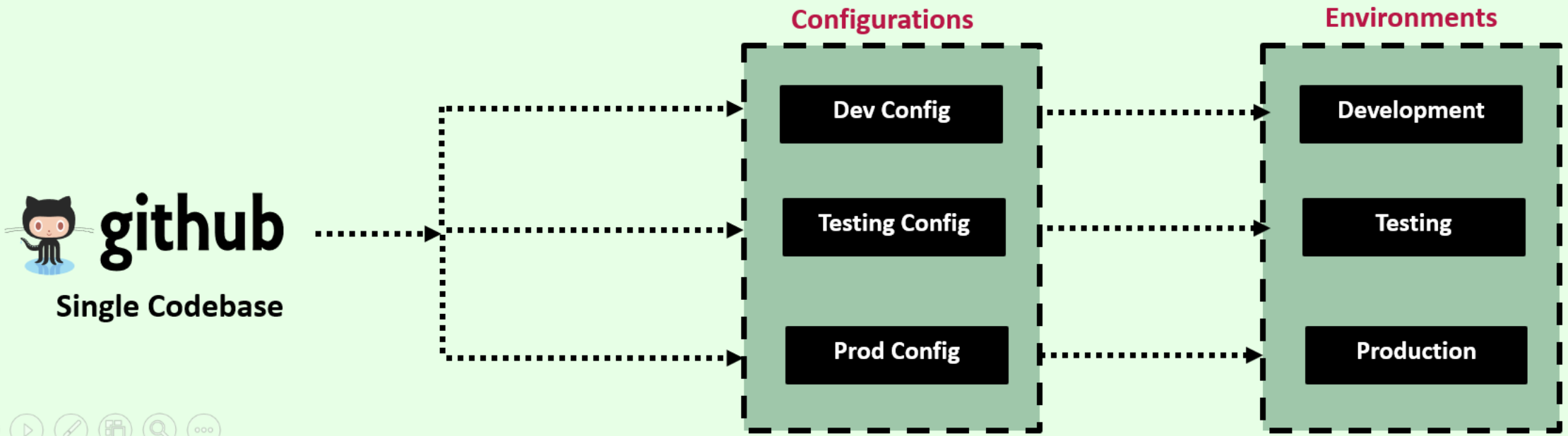
eazy bytes

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.

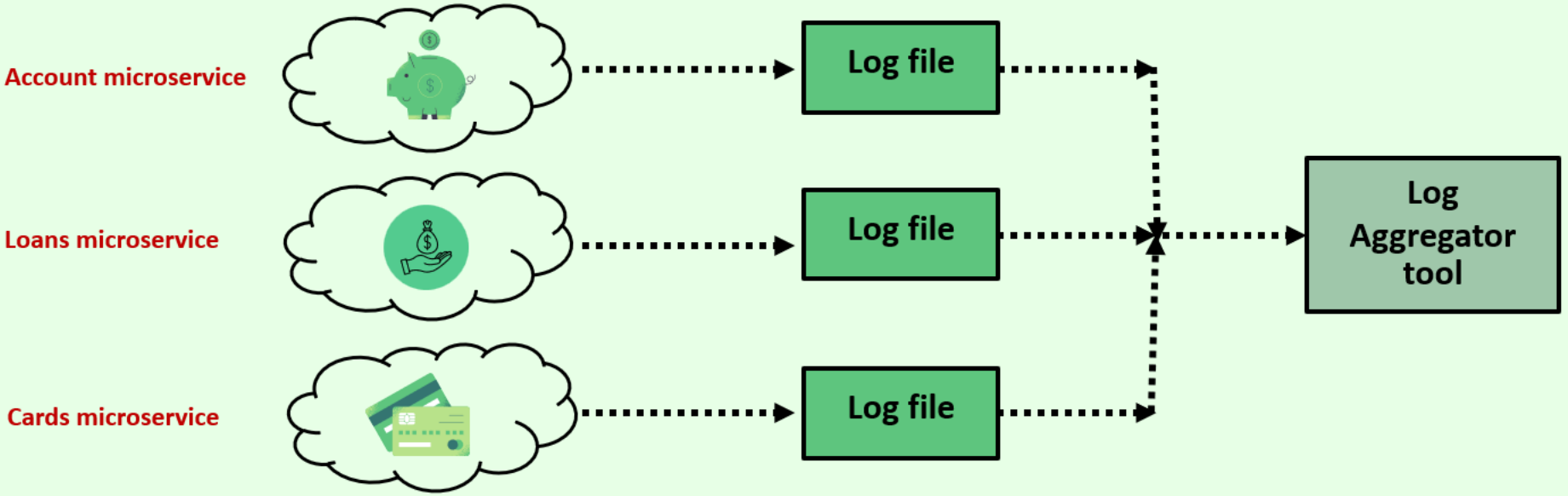
The methodology recommends utilizing environment variables to store configuration. This enables deploying the same application in different environments while adapting its behavior based on the specific environment's configuration.



15-Factor methodology - Logs

eazy bytes

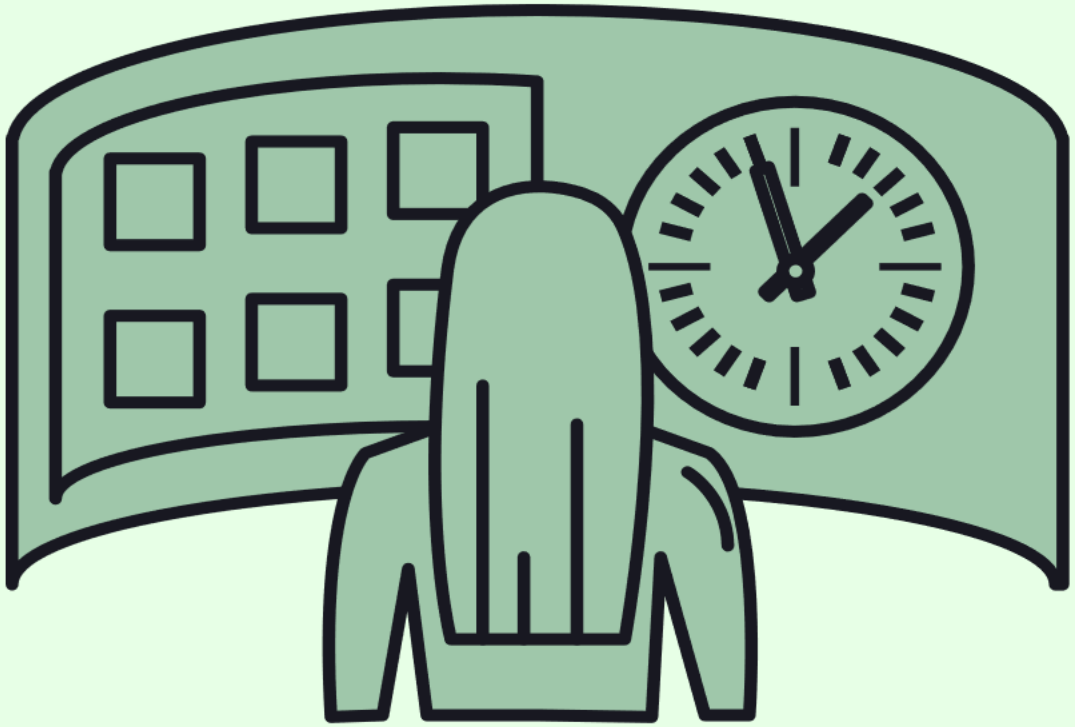
In a cloud-native application, log routing and storage are not the application's concern. Instead, applications should direct their logs to the standard output, treating them as sequentially ordered events based on time. The responsibility of log storage and rotation is now shifted to an external tool,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

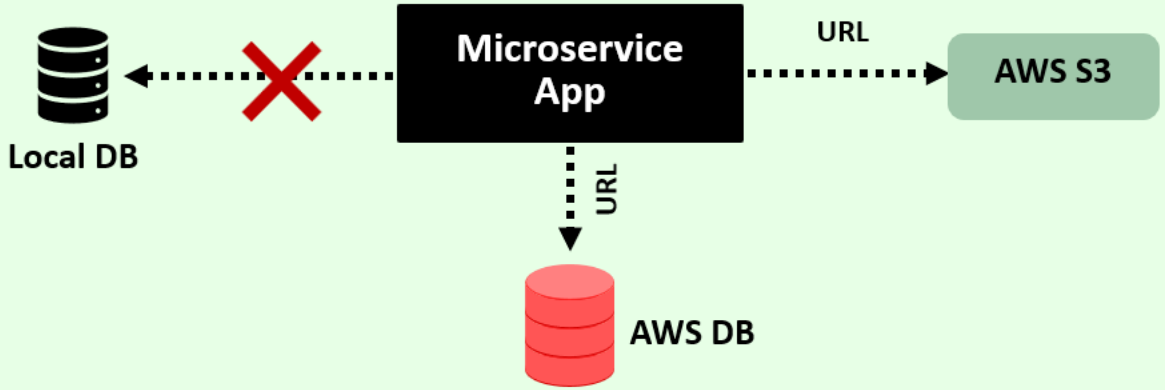
15-Factor methodology - Backing services

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

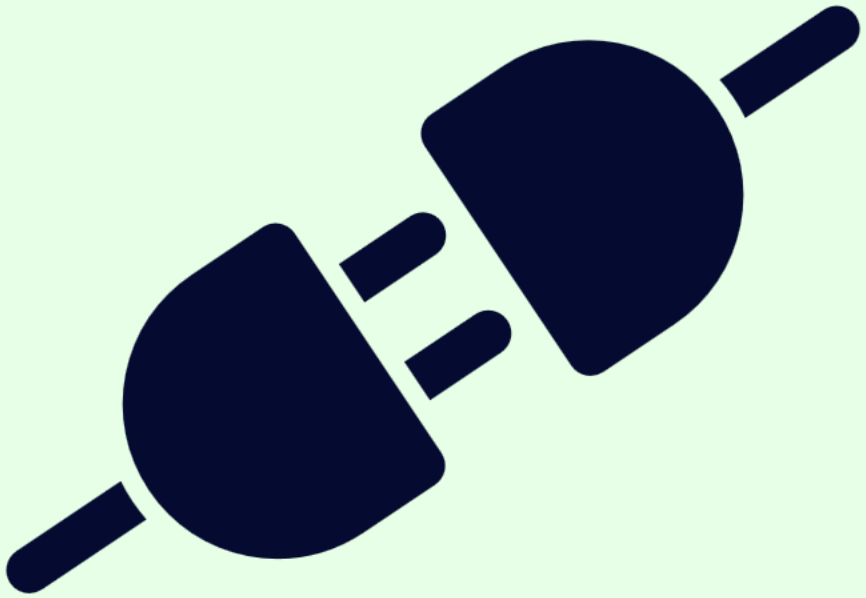
Management tasks required to support applications, such as database migrations, batch jobs, or maintenance tasks, should be treated as isolated processes. Similar to application processes, the code for these administrative tasks should be version controlled, packaged alongside the application, and executed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

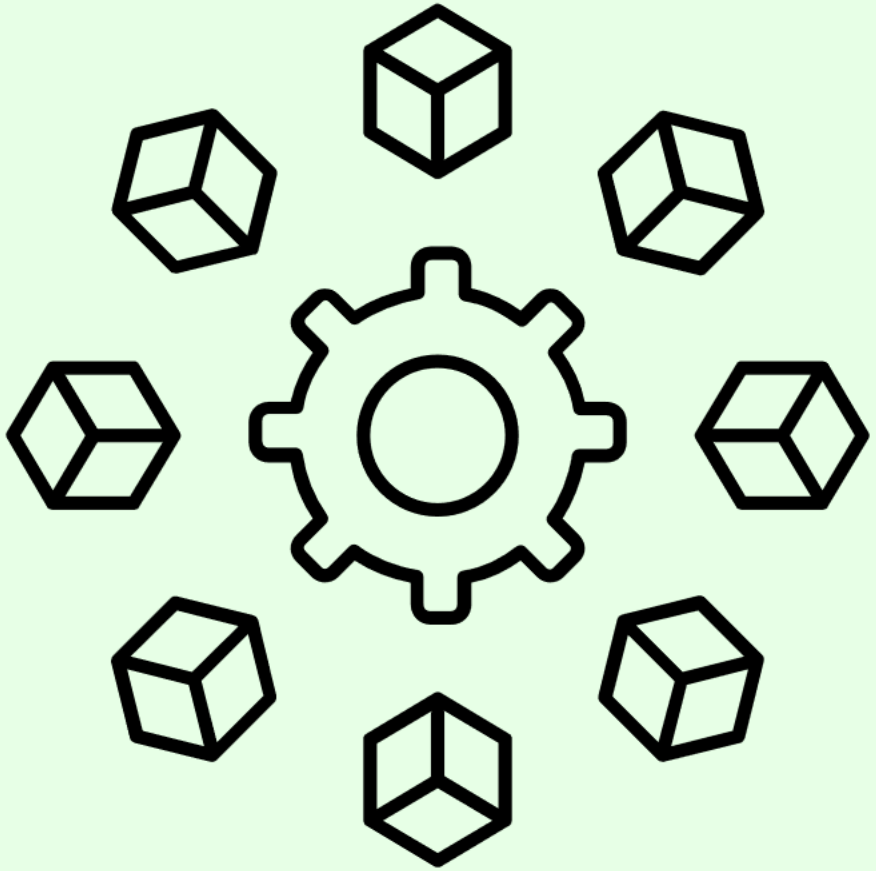
eazy bytes

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.

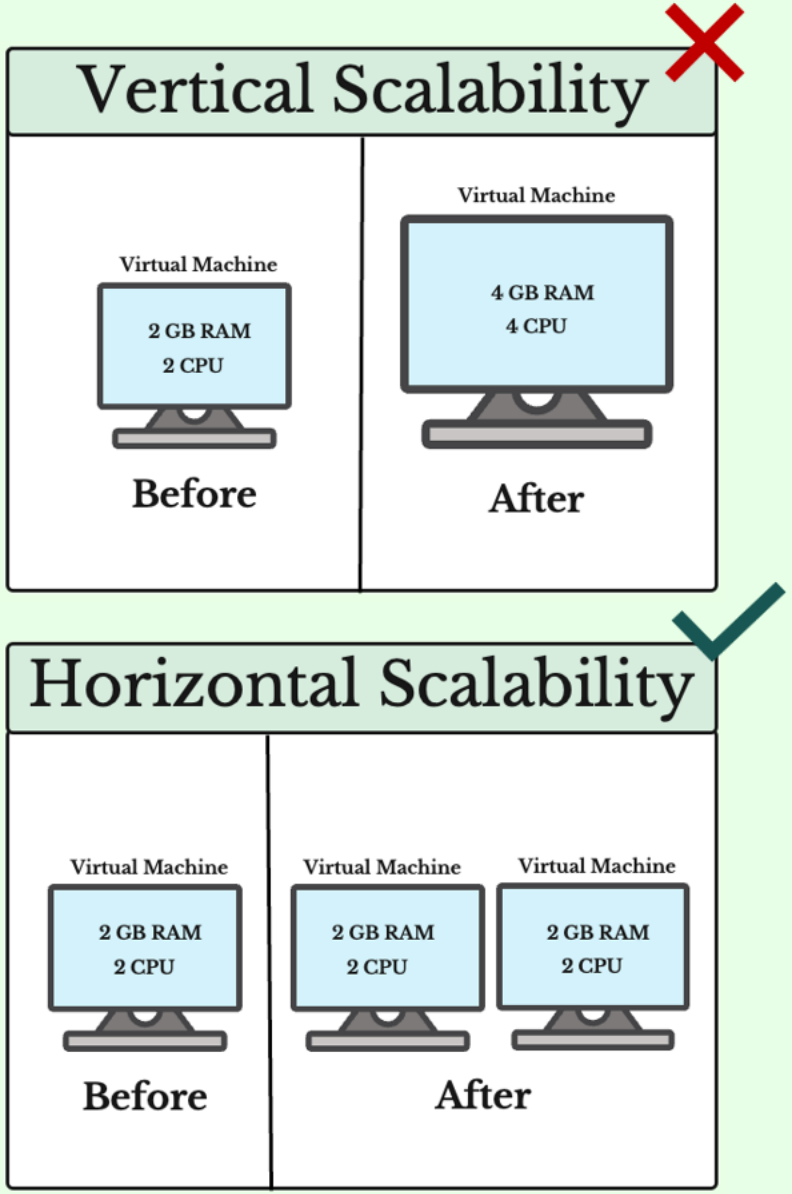
15-Factor methodology - Concurrency

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.



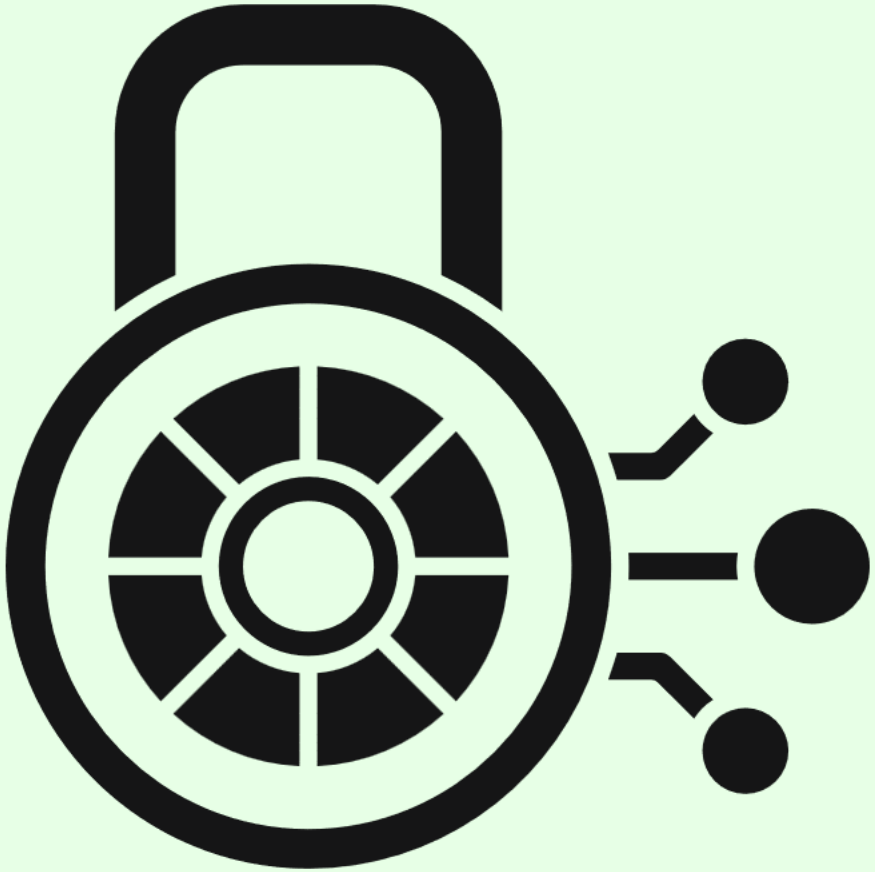
15-Factor methodology - Telemetry

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.

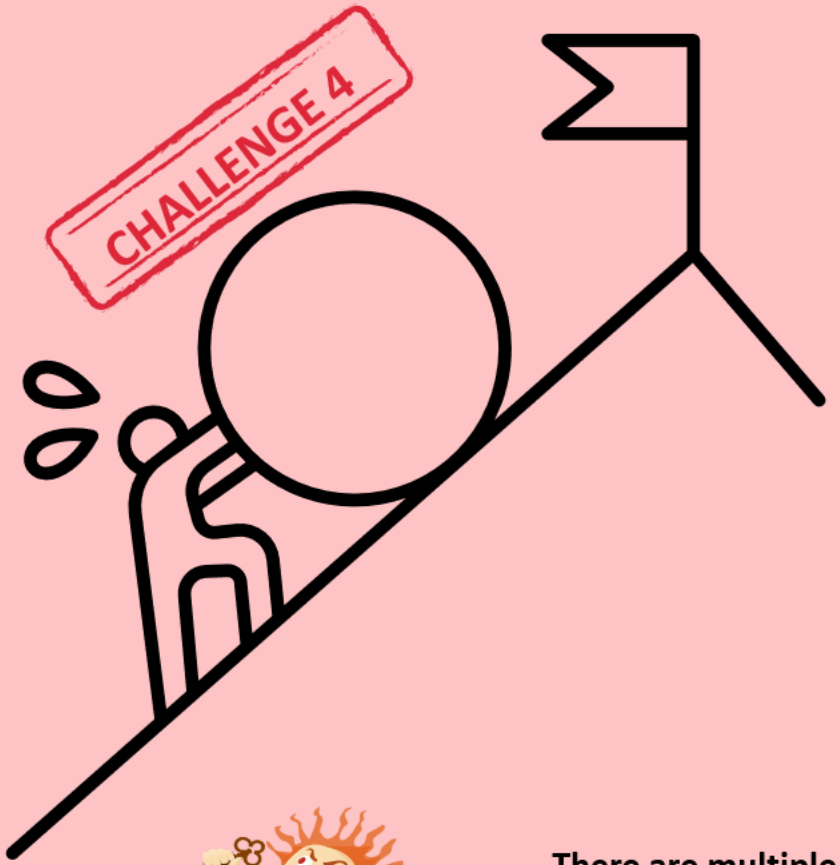


Security is a critical aspect of a software system, yet it often doesn't receive the necessary emphasis it deserves. To uphold a zero-trust approach, it is essential to ensure the security of every interaction within the system, encompassing architectural and infrastructural levels. While security involves 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

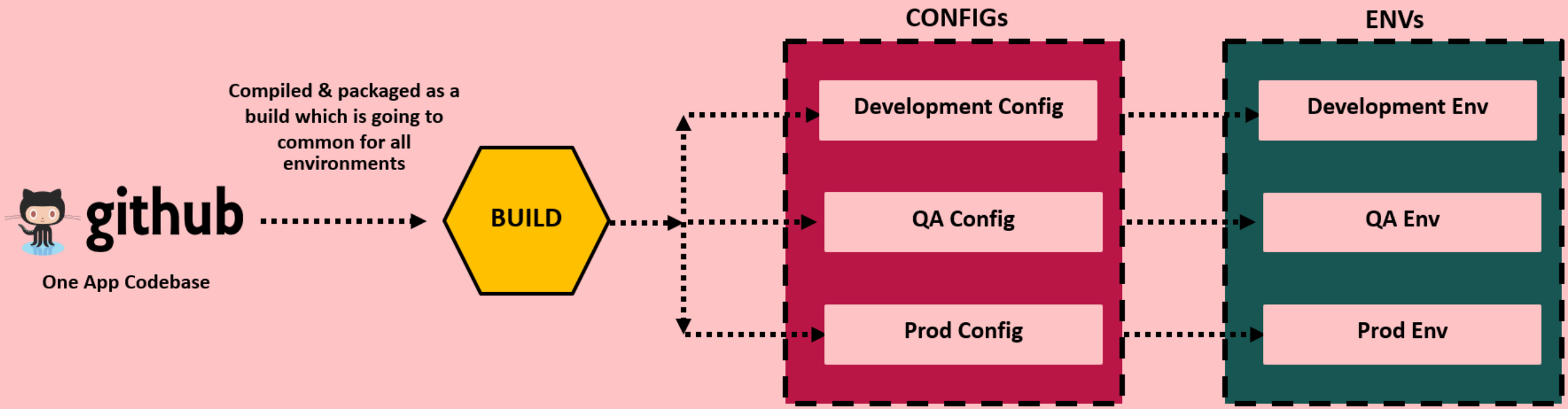
1) Configuring Spring Boot with properties and profiles
2) Applying external configuration with Spring Boot
3) Implementing a configuration server with Spring Cloud Config Server

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.

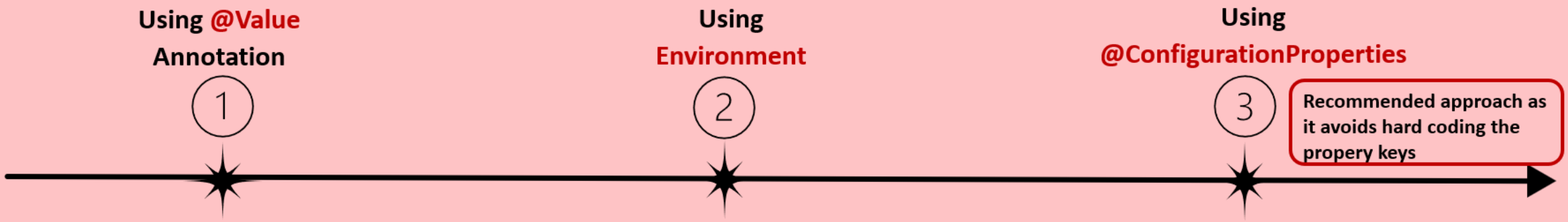
- ✓ Spring Boot uses a very particular order that is designed to allow sensible overriding of values. Properties are considered in the following order (with values from lower items overriding earlier ones):

- 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.

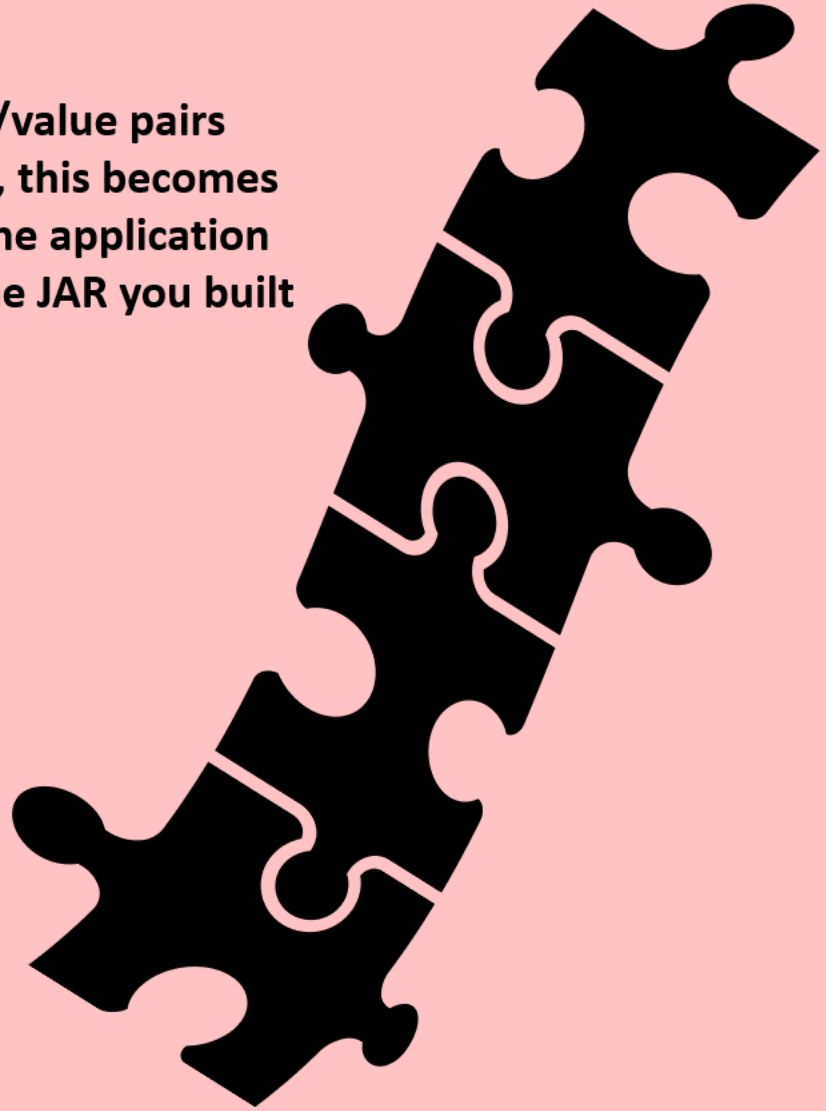
How to externalize configurations using command-line arguments ?

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?

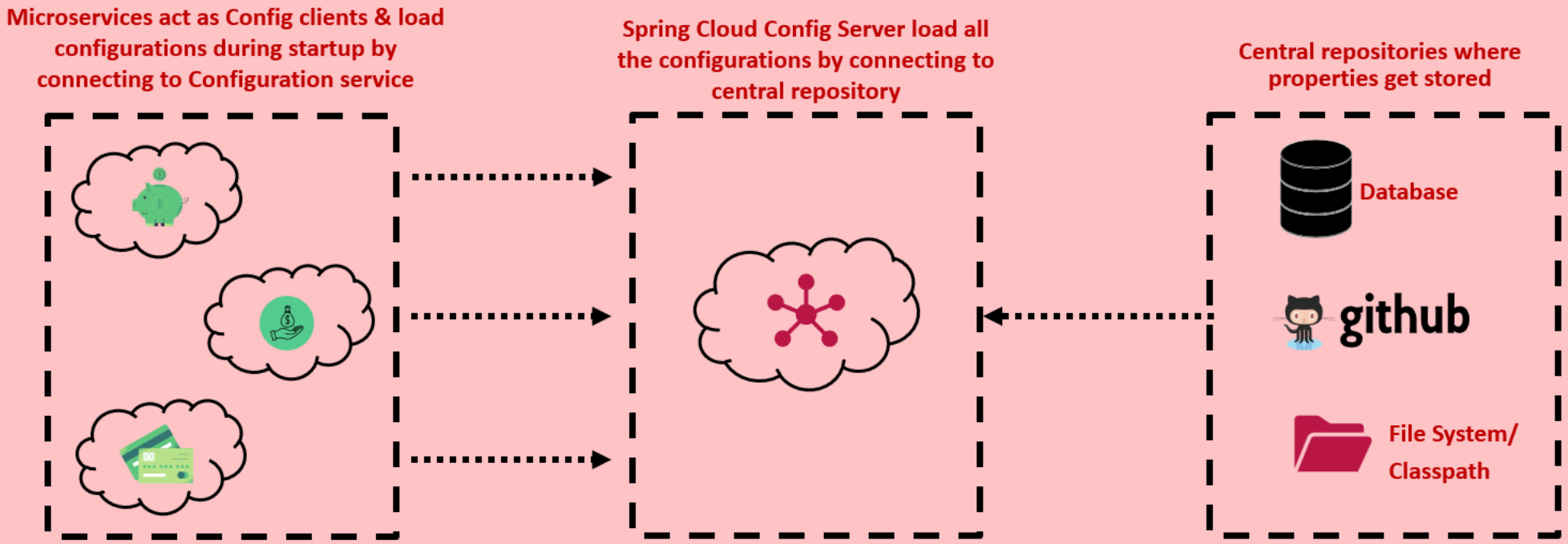
Spring Cloud Config

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,

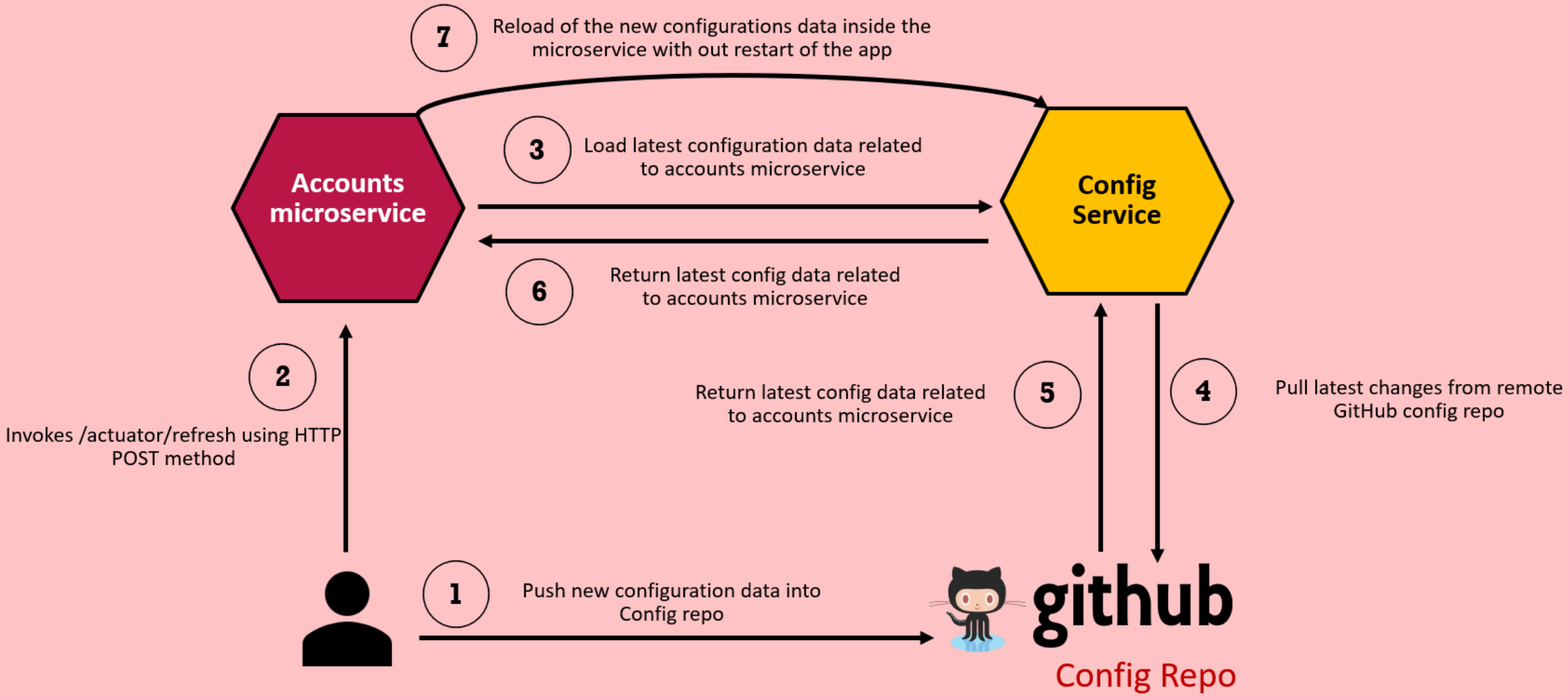
1. Add actuator dependency in the Config Client services: Add Spring Boot Actuator dependency inside pom.xml of the individual microservices like accounts, loans, cards to expose the /refresh endpoint

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,

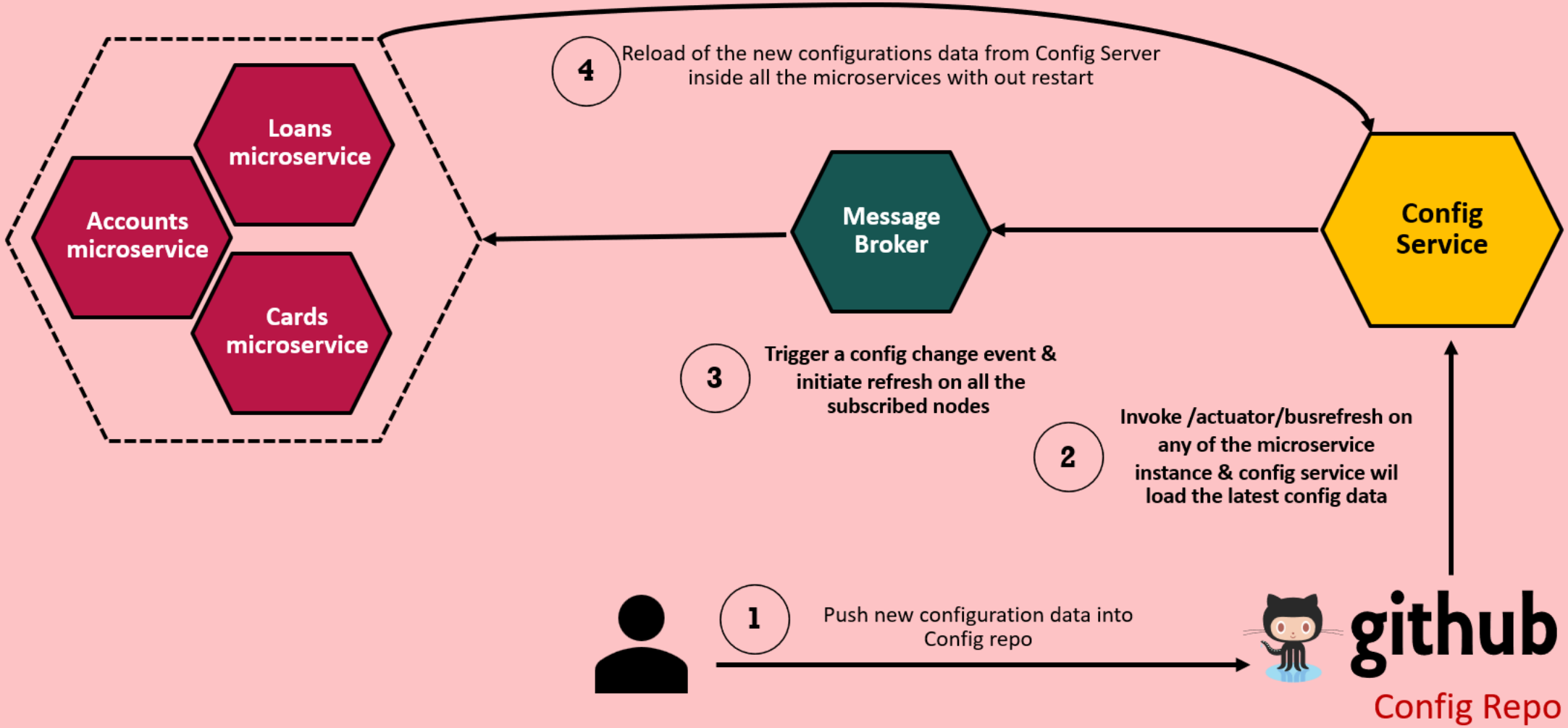
```
management:
  endpoints:
    web:
      exposure:
        include: busrefresh
```

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

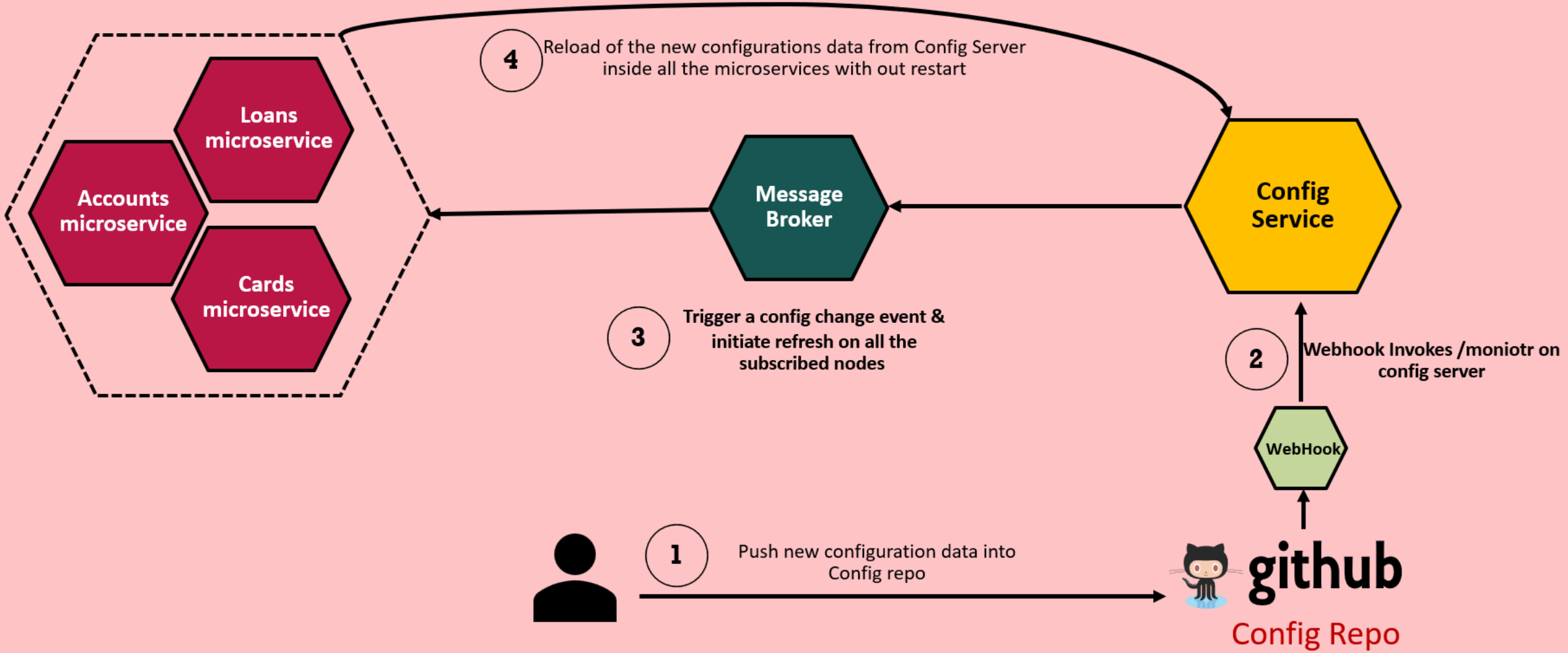
4. **Add Spring Cloud Config monitor dependency in the Config Server** : Add Spring Cloud Config monitor dependency (spring-cloud-config-monitor) inside pom.xml of Config server and this exposes /monitor endpoint

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



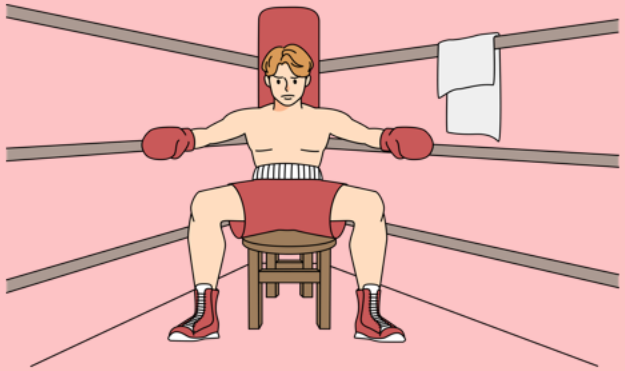
In this solution, there is no manual step involved and everything is automated.

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.

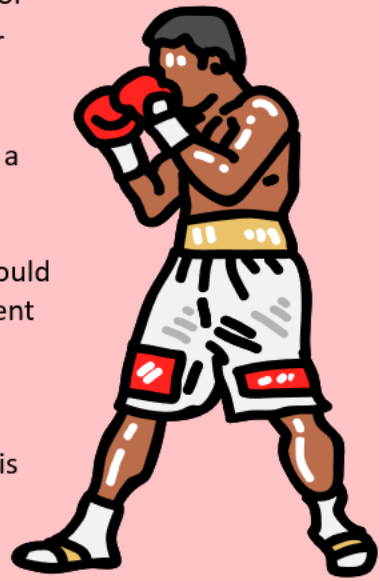
In simple words, liveness answers a true-or-false question: "Is this container alive?"



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.

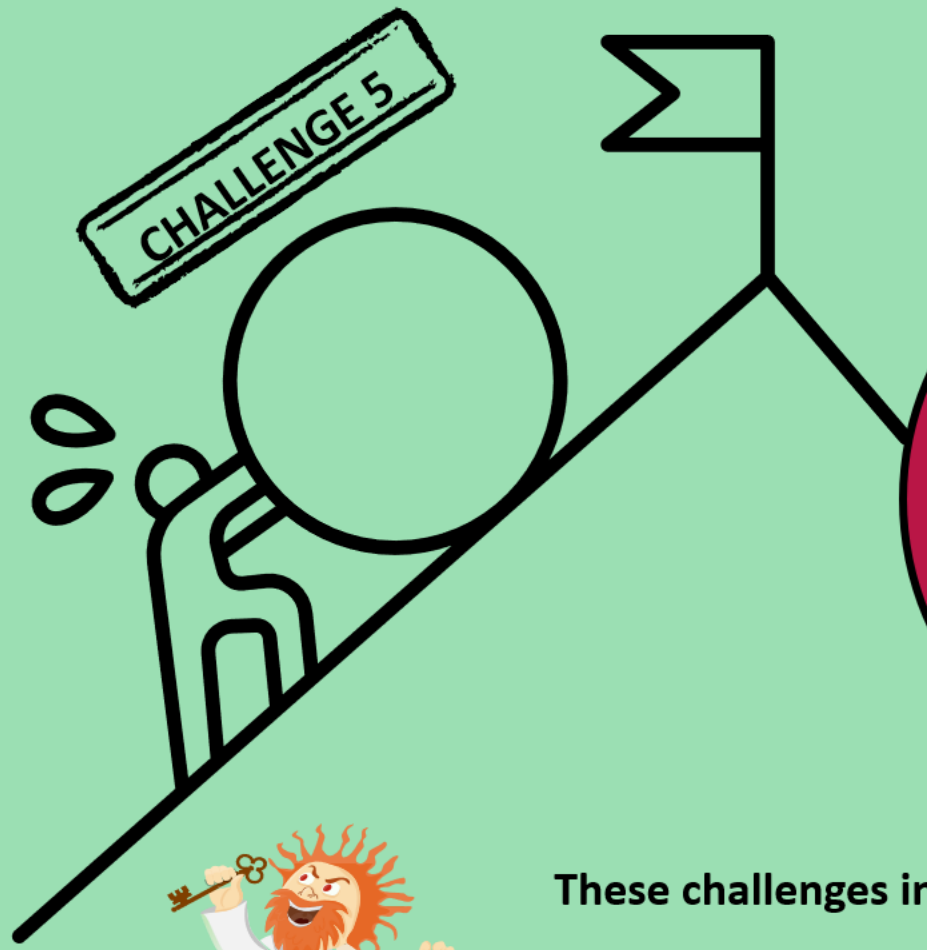
In simple words, readiness answers a true-or-false question: "Is this container ready to receive network traffic ?"



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?

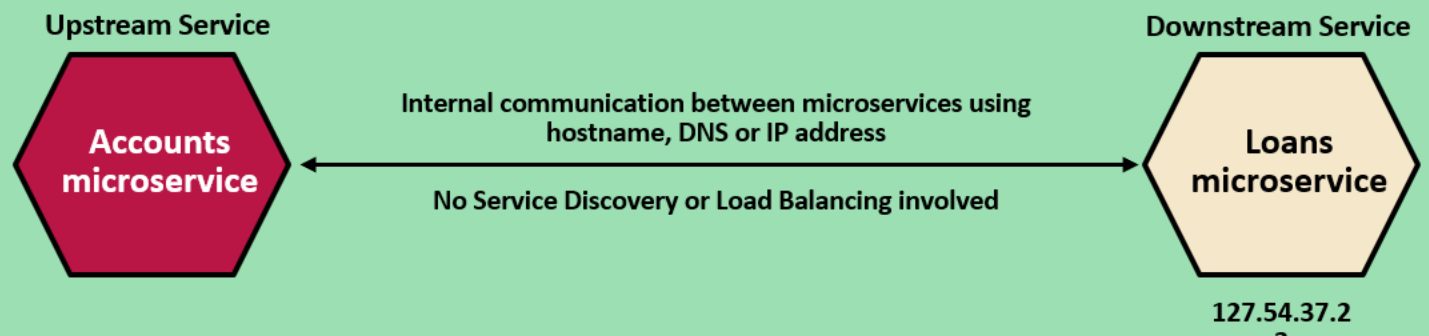
These challenges in microservices can be solved using below concepts or solutions,

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

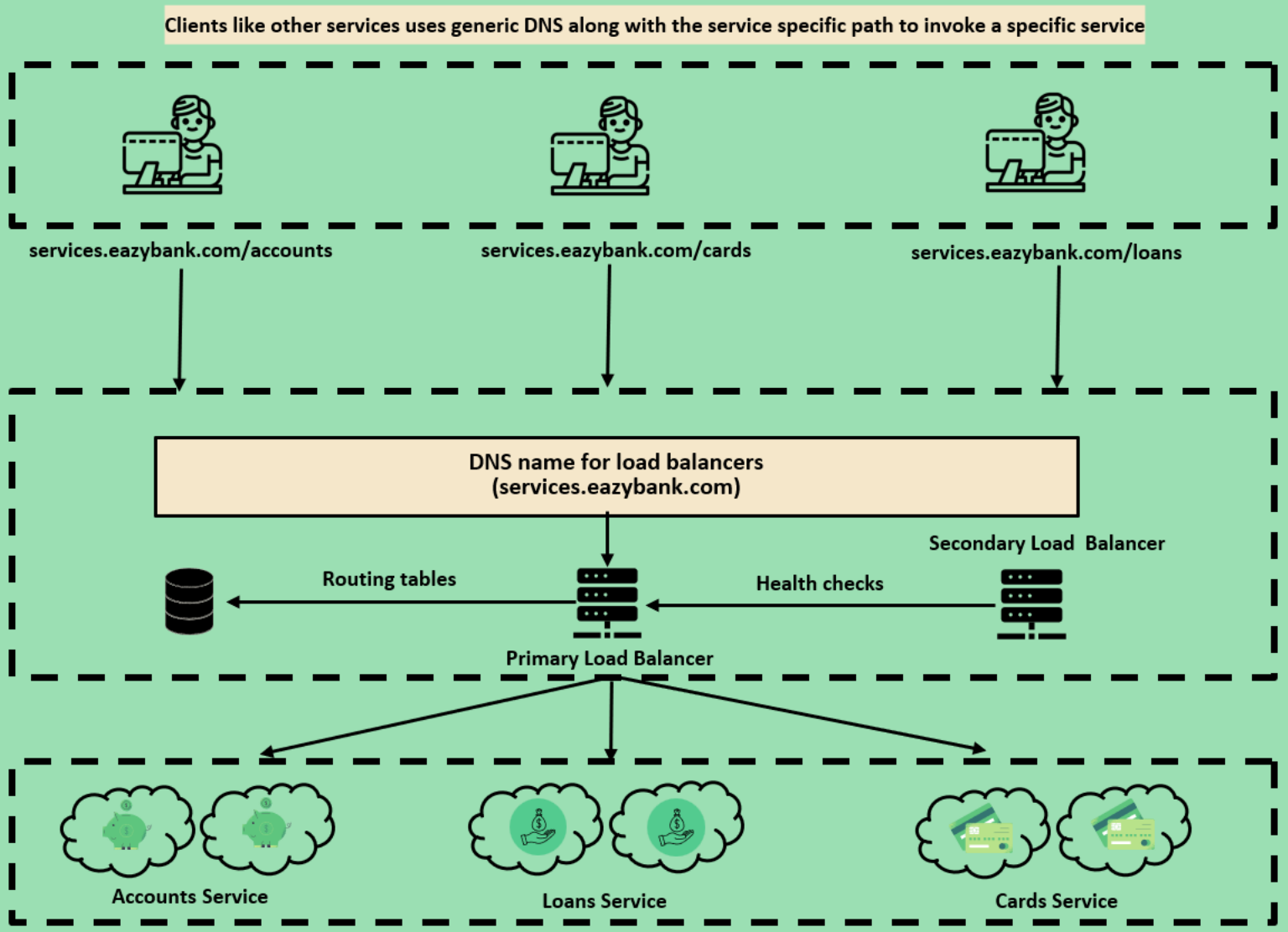
When there is only one instance of the Loans microservice running, managing the DNS name and its corresponding IP address mapping is straightforward. However, in a cloud environment, it is common to deploy multiple instances of a service, with each instance having its own unique IP address.

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,

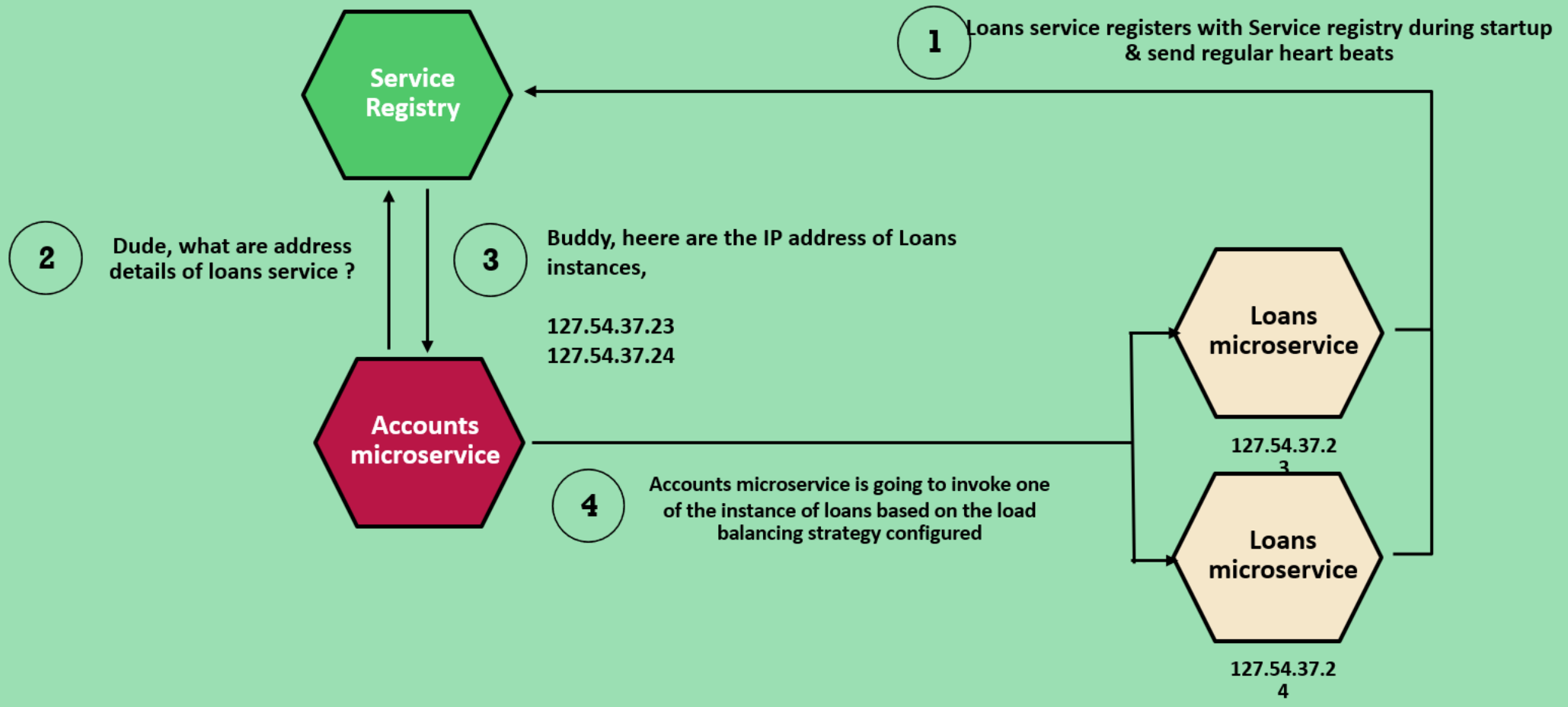
- A central server (or servers) that maintain a global view of addresses
- Microservices/clients that connect to the central server to register their address when they start & ready
- Microservices/clients need to send their heartbeats at regular intervals to central server about their health
- Microservices/clients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

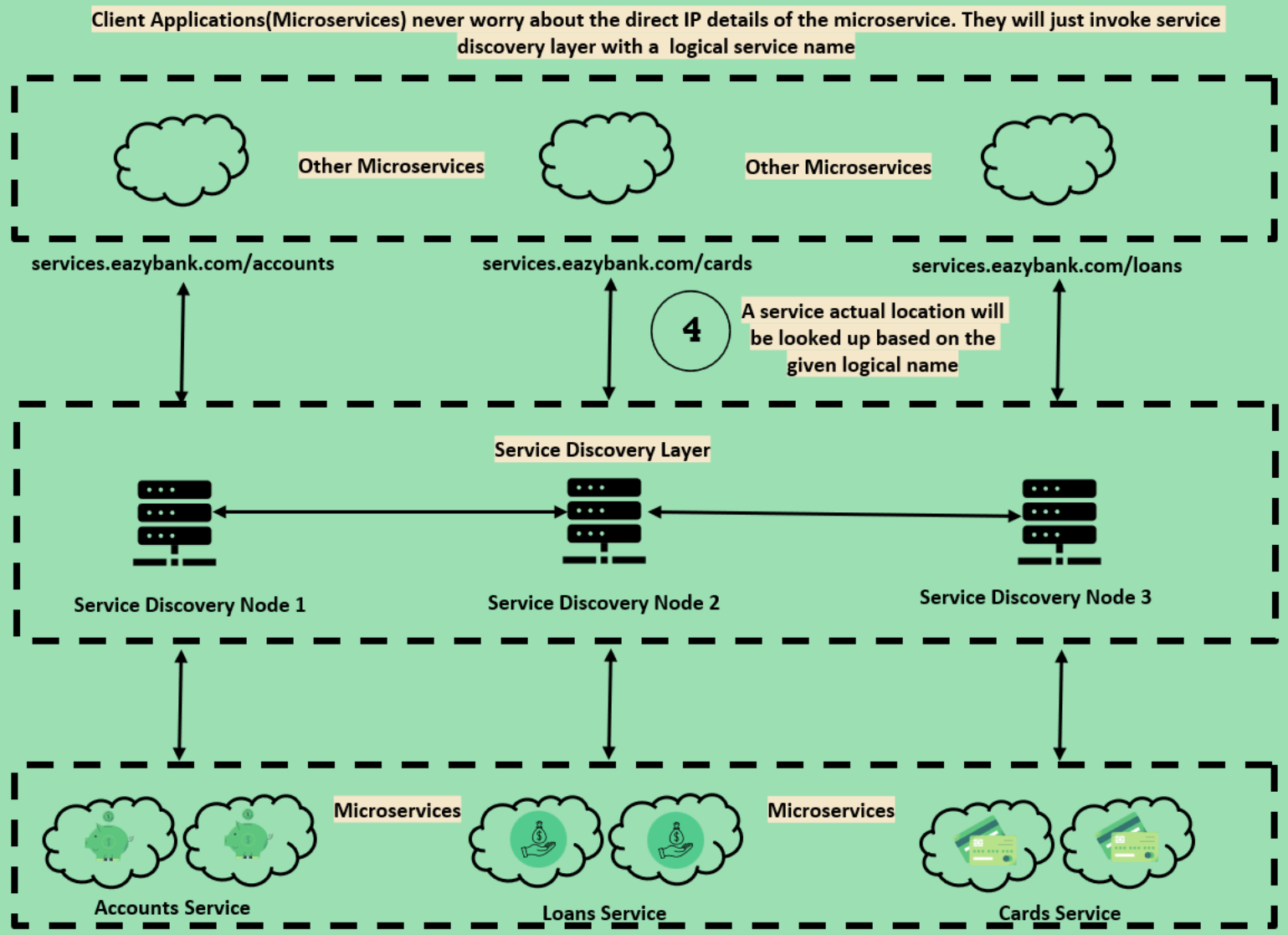
- **Service Registration:** Client applications register themselves with the service registry upon startup. They provide essential information about their location, such as IP address, port, and metadata, which helps identify and categorize the service.
- **Service Discovery:** When a client application 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

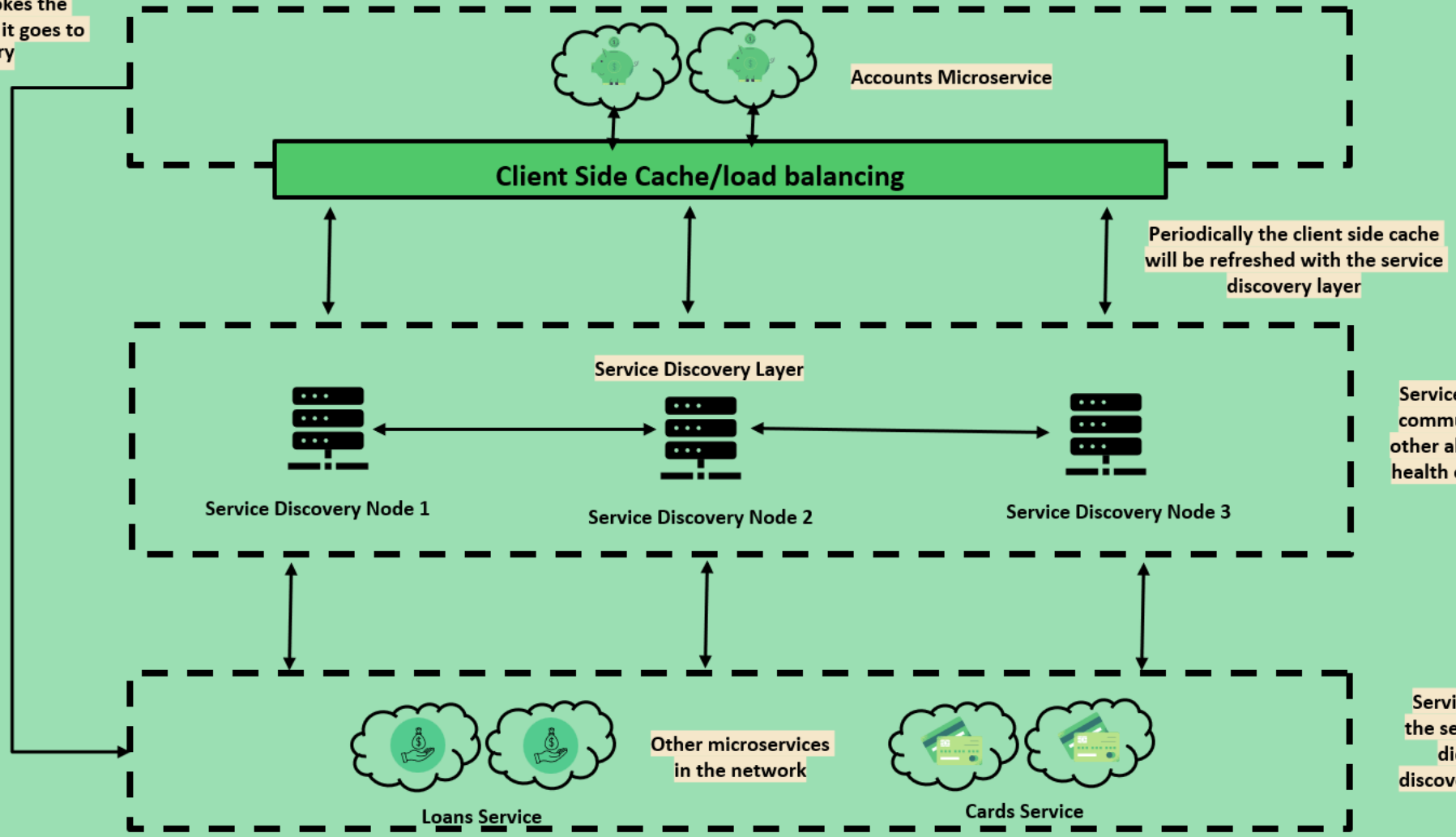


How loadbalancing works in Client-side service discovery ?

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

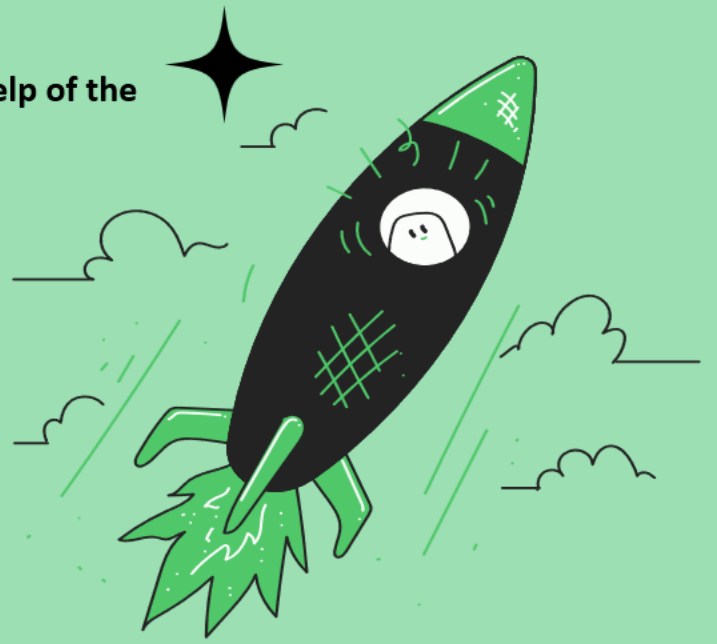


Spring Cloud support for Client-side service discovery

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.

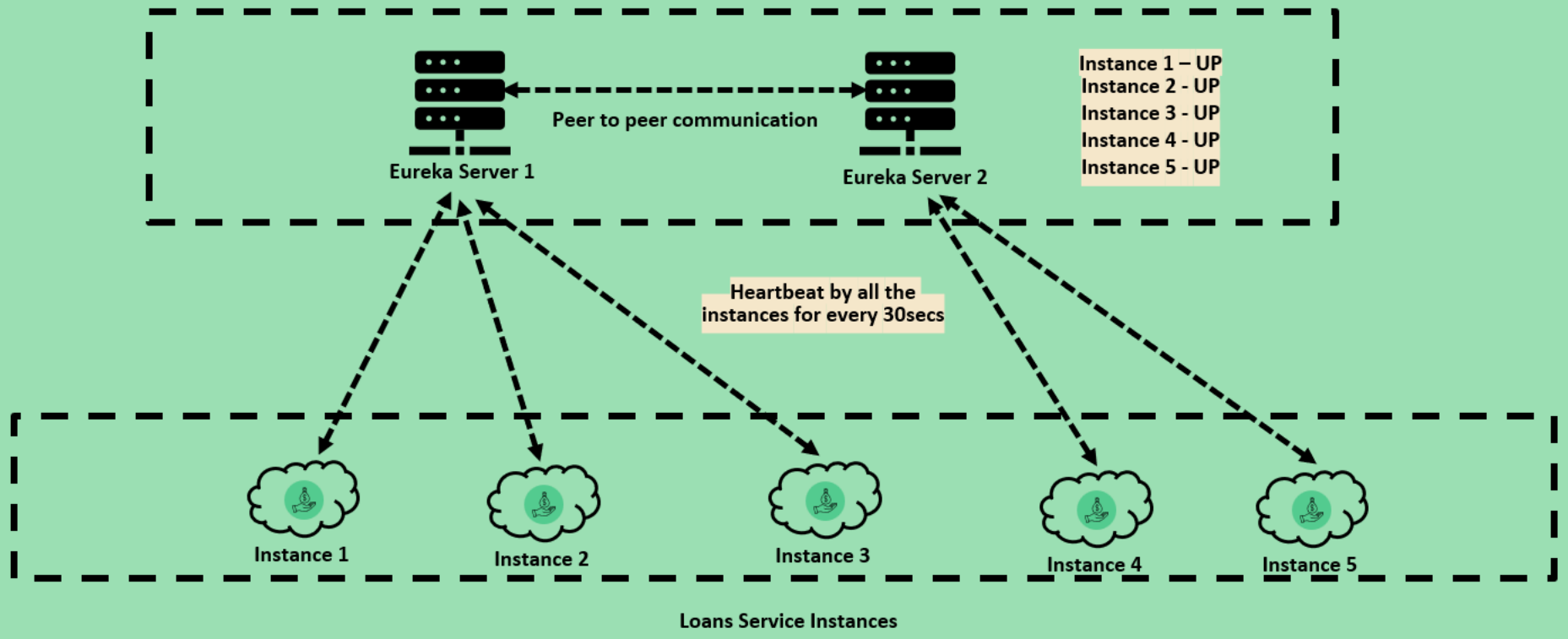
In Self-Preservation mode, the Eureka server continues to serve the registered instances to client applications, even if it suspects that some instances are no longer available. This helps maintain the stability and availability of the service registry, ensuring that clients can still discover and interact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

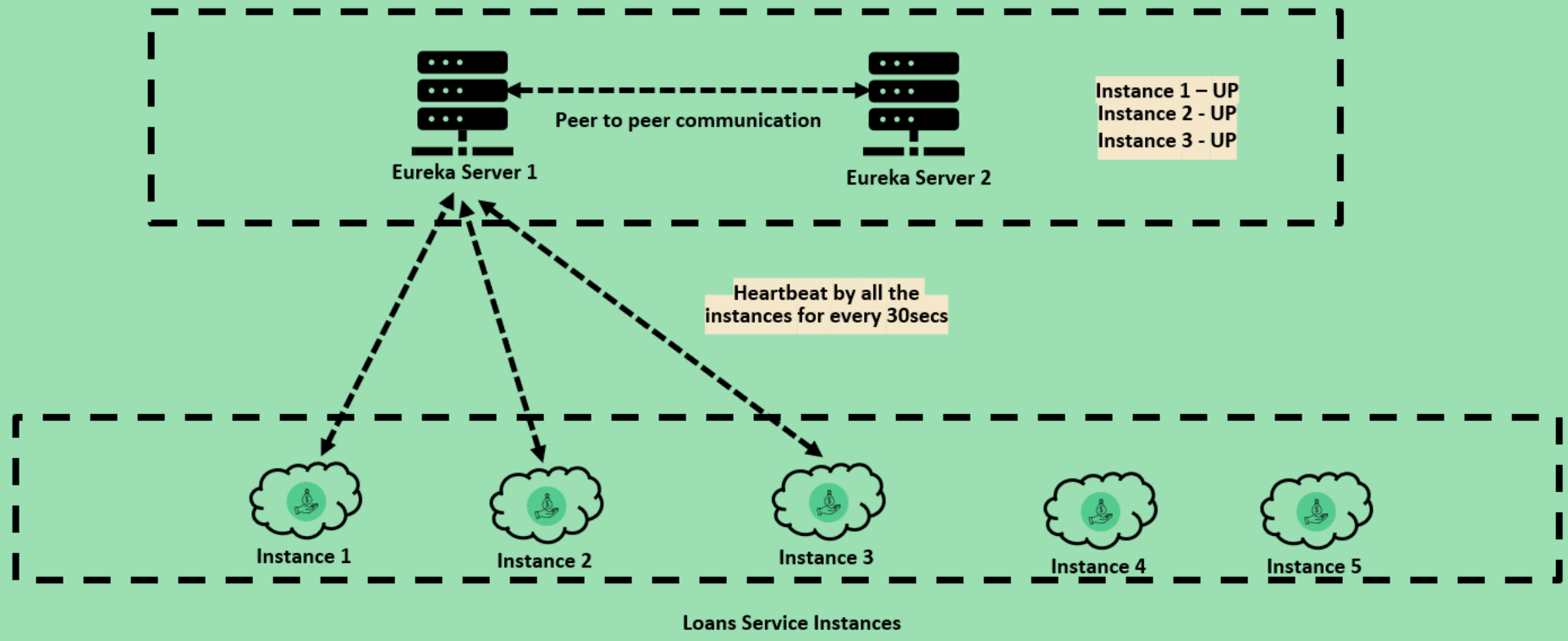
eazy bytes



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

EUREKA SELF-PRESERVATION TO AVOID TRAPS IN NETWORK

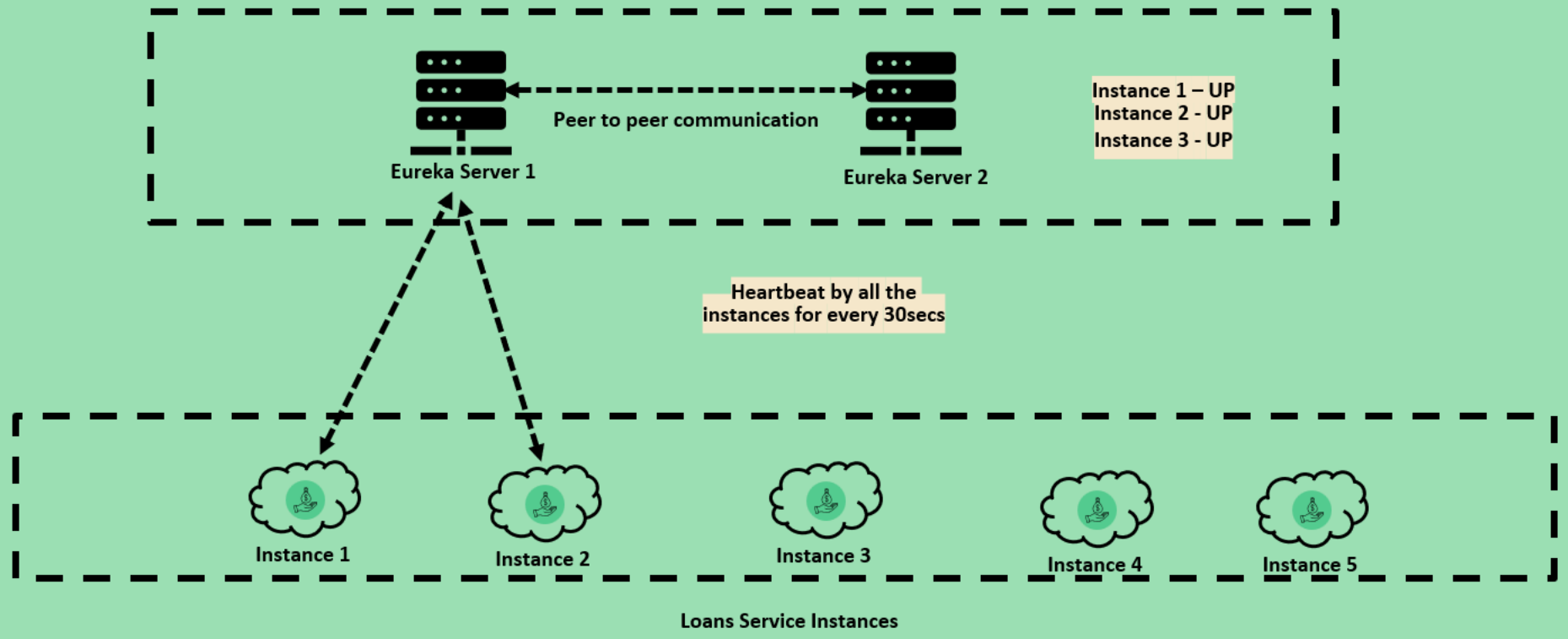
eazy bytes



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.

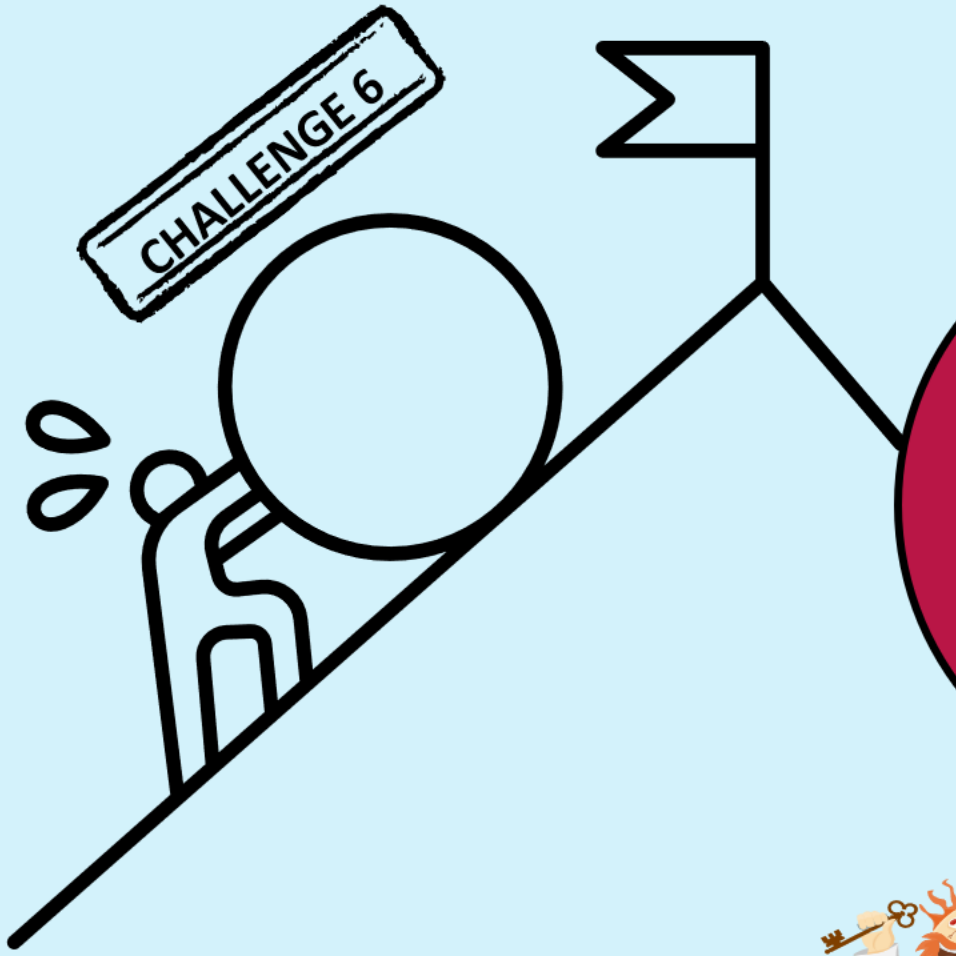
✓ `eureka.server.renewal-percent-threshold = 0.85`
This value is used to calculate the expected % of heartbeats per minute eureka is expecting.

✓ `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

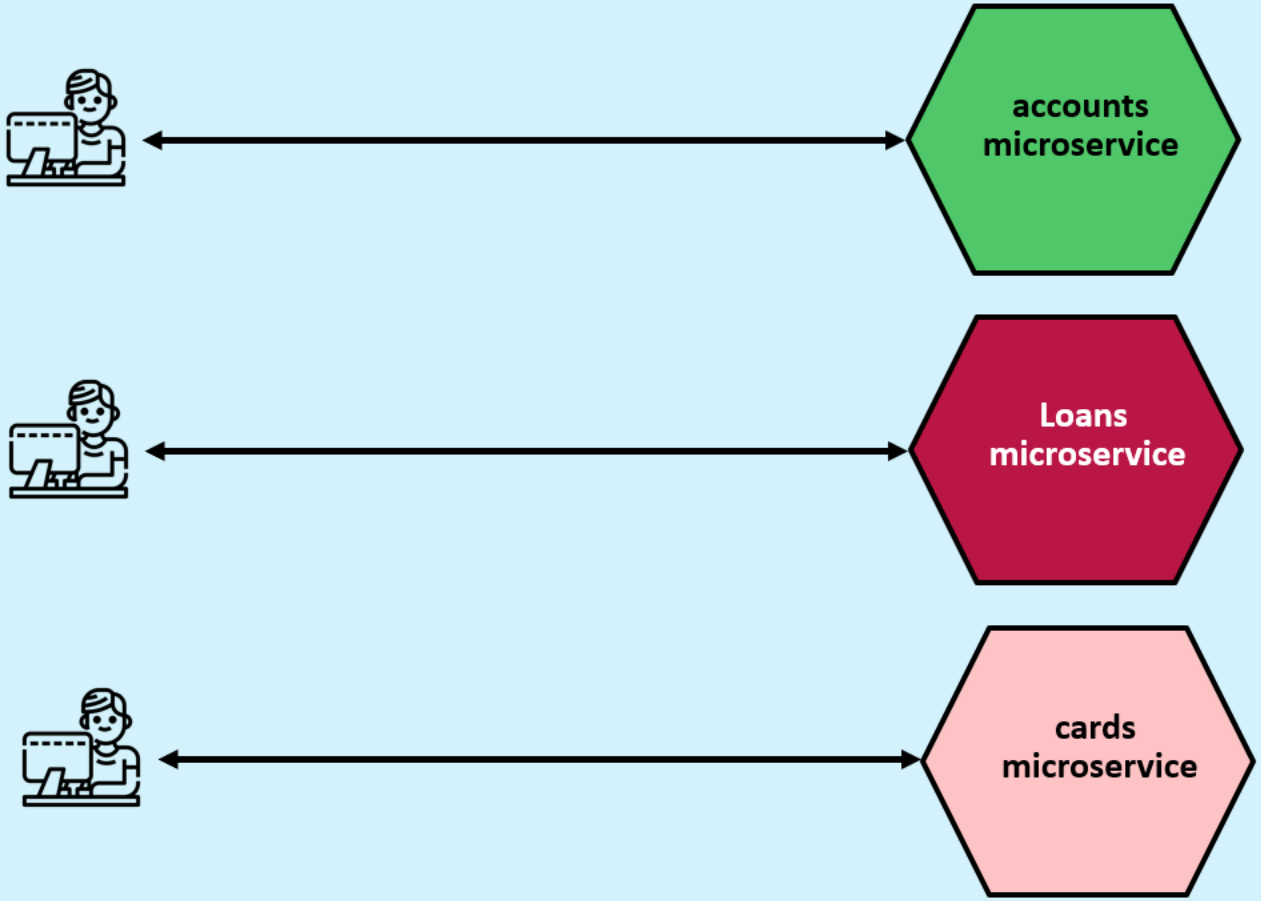
How to provide dynamic routing capabilities which allows to define routing rules based on various criteria such as HTTP headers, request parameters etc. inside microservices network ?

These challenges in microservices can be solved using a

Edge server

ROUTING, CROSS CUTTING CONCERNS IN MICROSERVICES

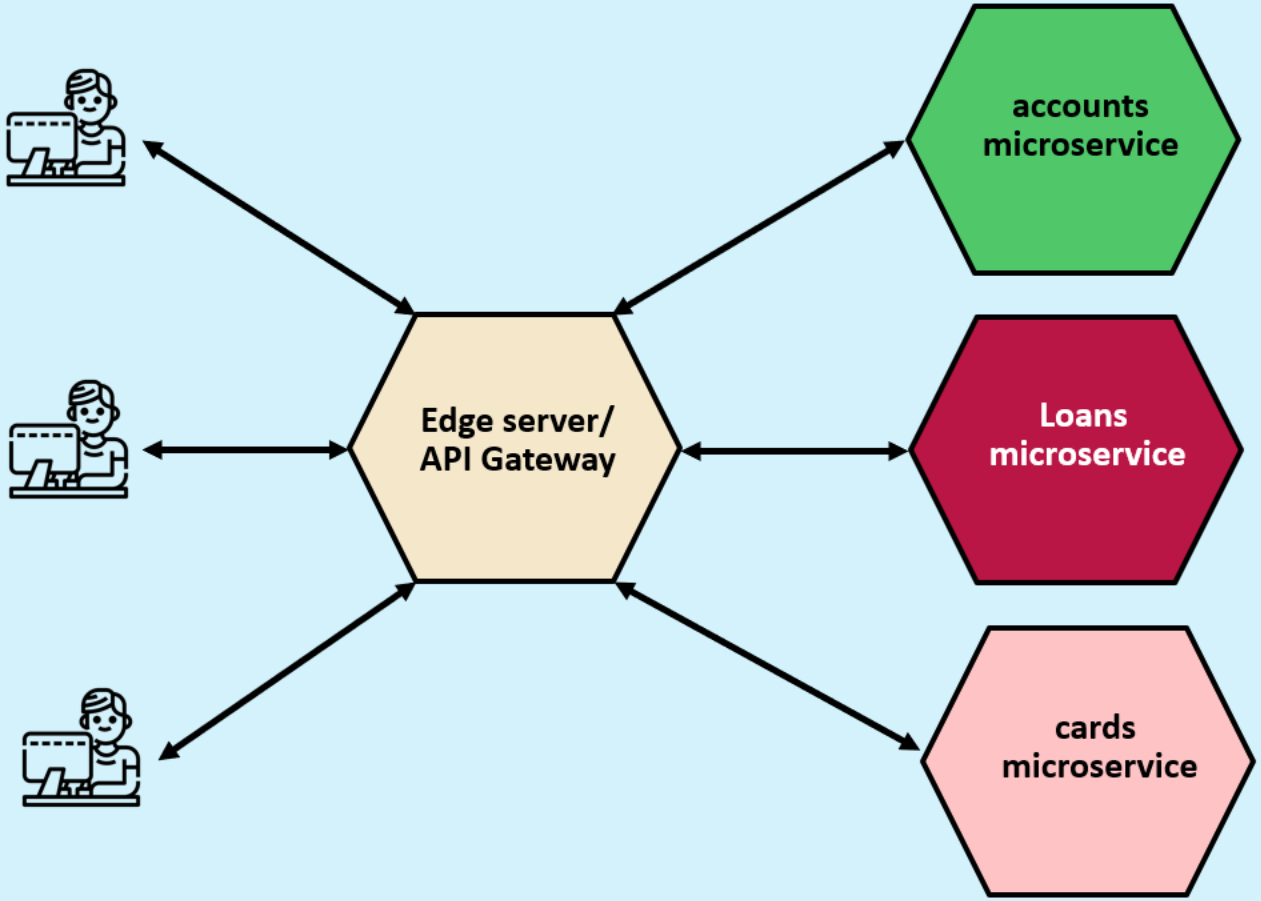
eazy bytes



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

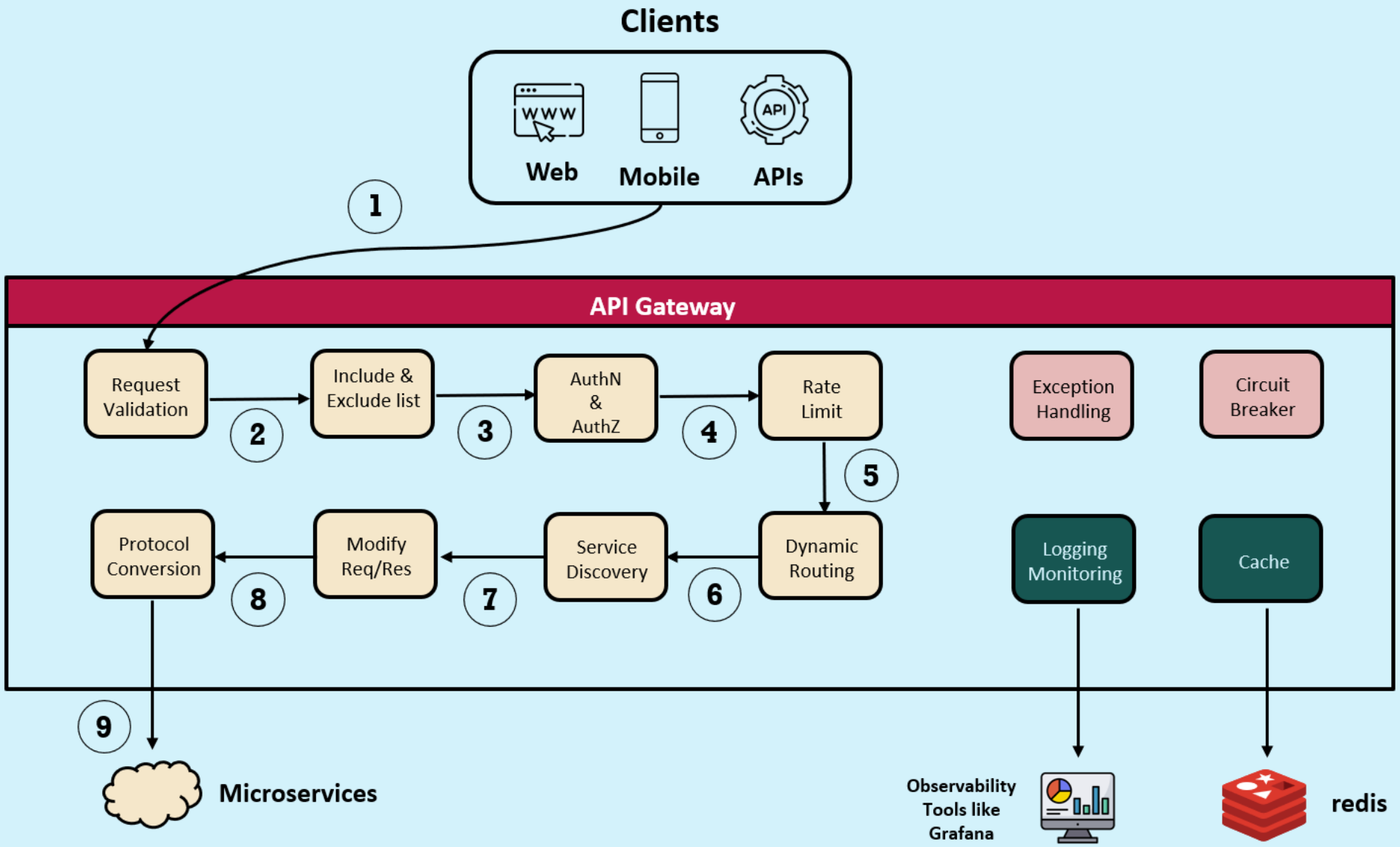
eazy bytes



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.

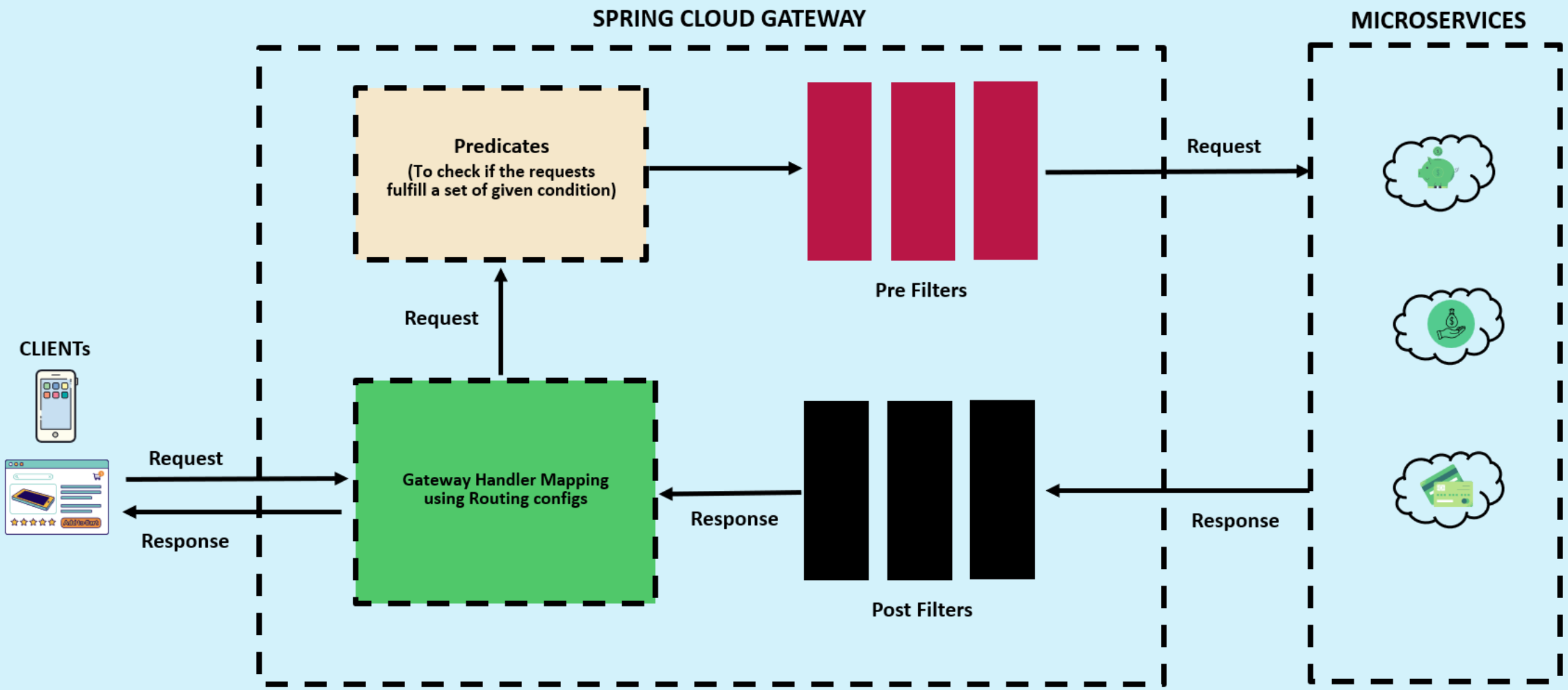
The service gateway sits between all calls from the client to the individual services & acts as a central Policy Enforcement Point (PEP) like below,

- 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,

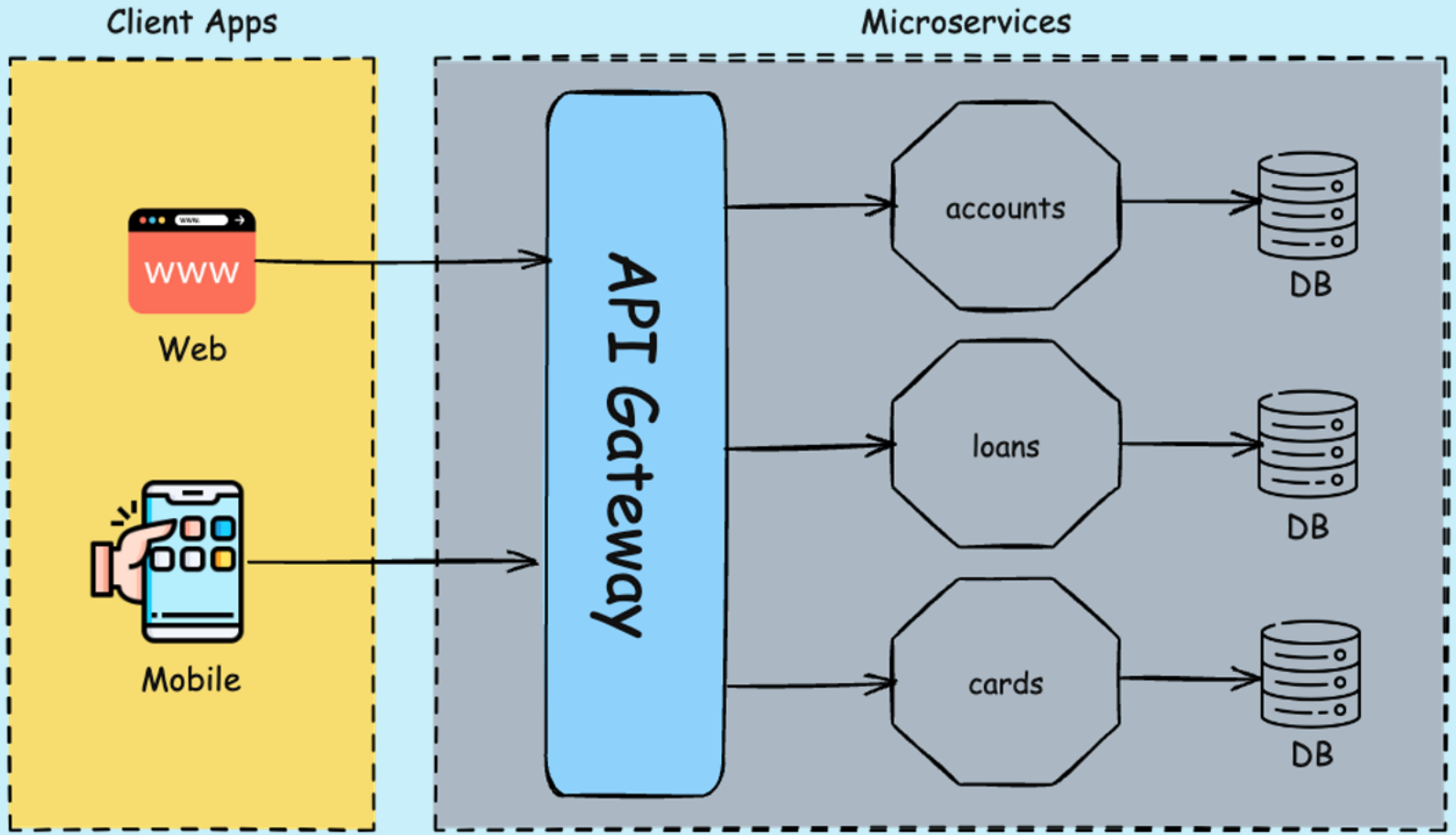
```
@Bean
public RouteLocator myRoutes(RouteLocatorBuilder builder) {
        return builder.routes()
                .route(p -> p
                        .path("/eazybank/accounts/**")
                        .filters(f -> f.rewritePath("/eazybank/accounts/(?<segment>.*)","/${segment}")
          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

API Gateway Pattern

eazy bytes

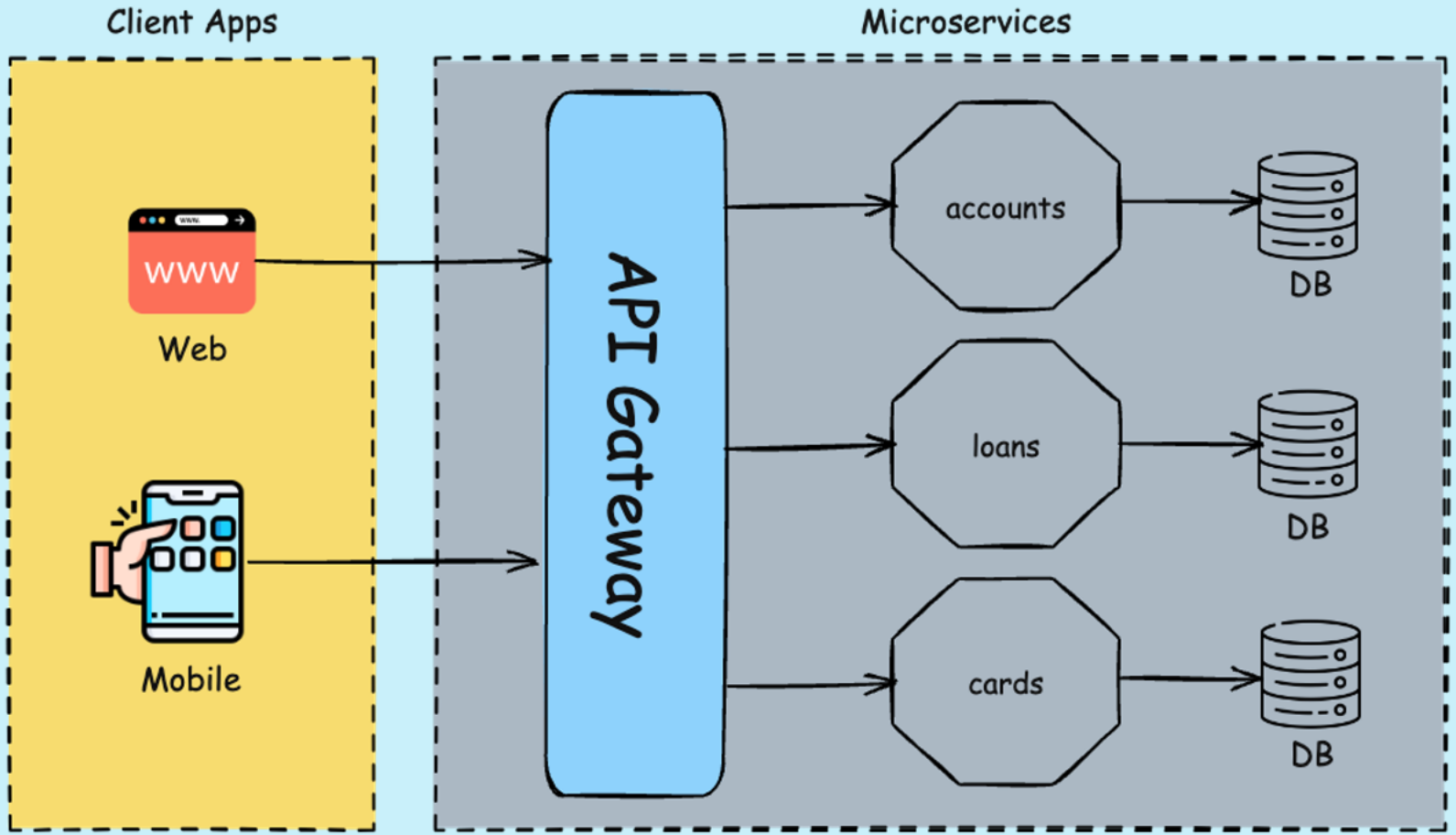
The API Gateway Pattern is a critical architectural component in microservices design, offering a unified entry point for multiple microservices. It acts as a gateway between the external clients (e.g., web apps, mobile apps) and the internal microservices, helping streamline communication, security, and routing. This pattern is essential when managing the complexities of microservice-based applications.



Gateway Routing pattern

eazy bytes

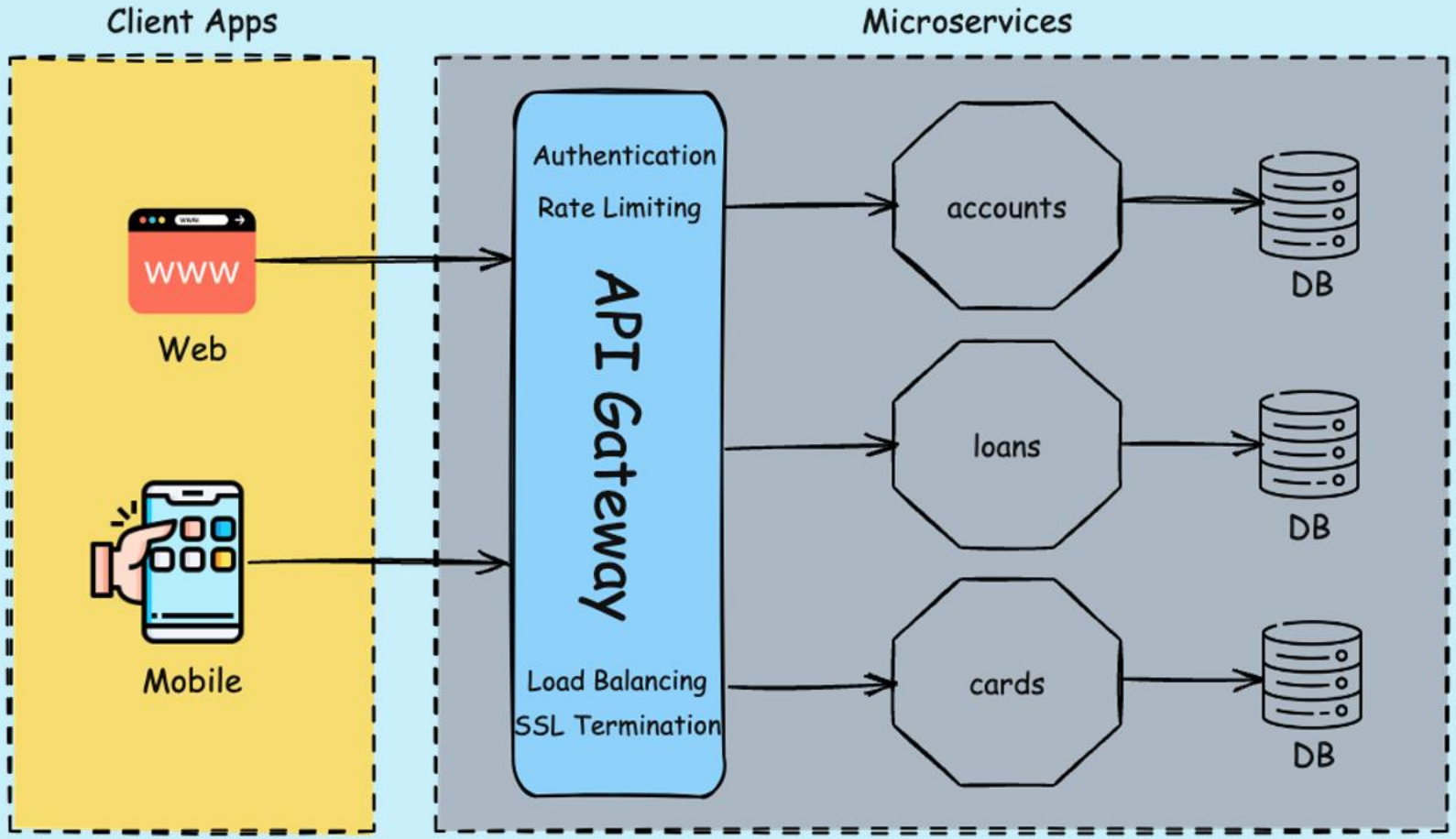
The Gateway Routing pattern is a design pattern used in microservices architectures where an API Gateway routes incoming client requests to the appropriate backend microservices based on various factors like the URL, headers, or request parameters.



Gateway offloading Pattern

eazy bytes

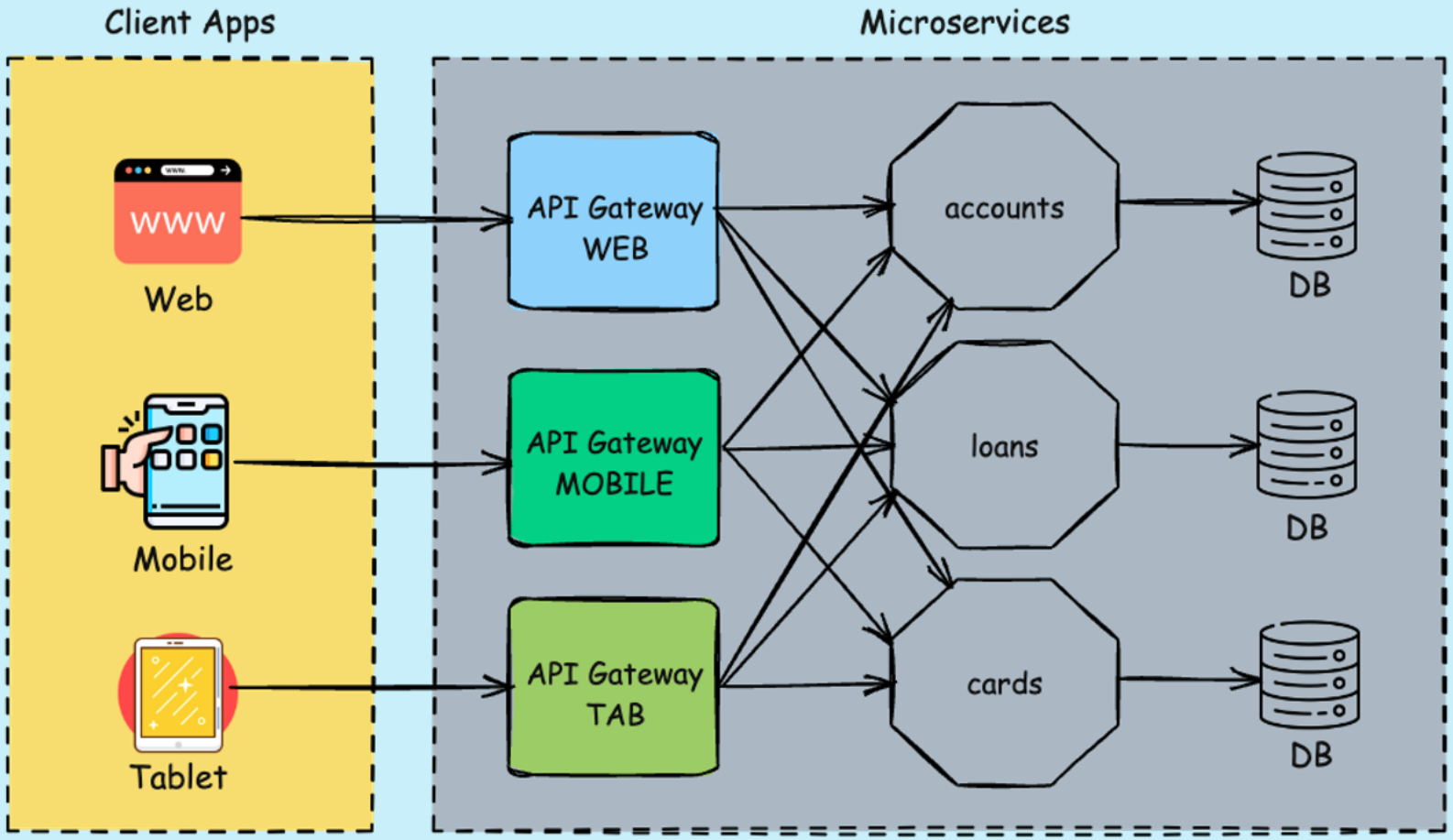
The Gateway Offloading Pattern is an architectural pattern used in microservices to offload certain cross-cutting concerns—such as security, caching, rate limiting, and monitoring—from individual microservices to the API Gateway. This pattern helps centralize and simplify the implementation of these concerns, allowing the microservices to focus solely on business logic.



Backend for Frontend (BFF) Pattern

eazy bytes

The Backend for Frontend (BFF) Pattern is a design pattern used in microservices architectures where a separate backend service is created for each client type (e.g., web, mobile, tablet). Each frontend (client) has its own specialized backend to optimize communication between the frontend and the microservices, providing a tailored experience for different clients.



Gateway Aggregator/Composition pattern

eazy bytes

In microservices architecture, a Gateway Aggregator or Gateway Composition pattern is used when a request from a client needs to retrieve or process data from multiple backend microservices. Instead of having the client make multiple calls to various microservices, the API Gateway consolidates the requests into a single response.