RESILIENCY IN MICROSERVICES

eazy bytes

HOW DO WE AVOID CASCADING FAILURES?

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

HOW DO WE HANDLE FAILURES GRACEFULLY WITH FALLBACKS?

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

HOW TO MAKE OUR SERVICES SELF-HEALING CAPABLE

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

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

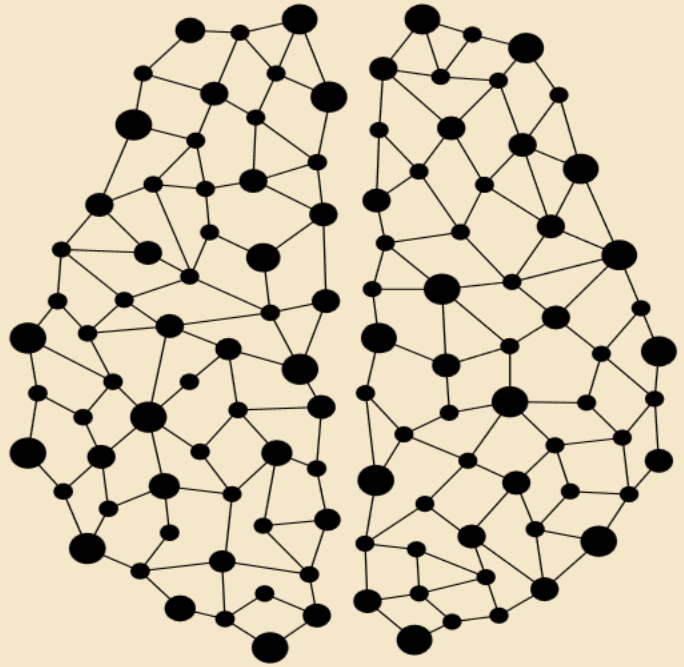
There exist various patterns for building resilient applications. In the Java ecosystem, Hystrix, a library developed by Netflix, was widely used for implementing such patterns. However, Hystrix entered maintenance mode in 2018 and is no longer being actively developed. To address this, Resilience4J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

RESILIENCY USING RESILIENCE4J

eazy bytes

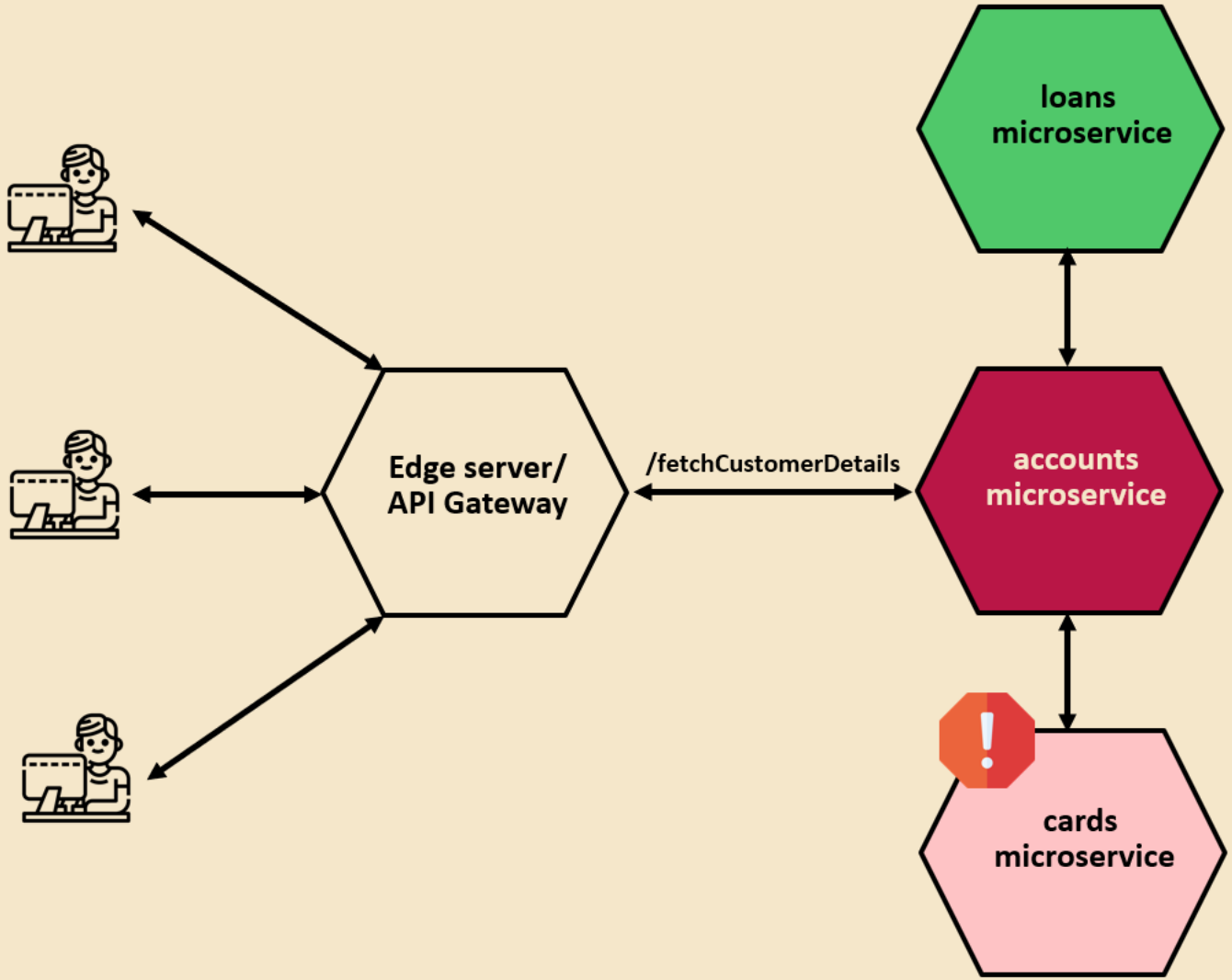
Resilience4j is a lightweight fault tolerance library designed for functional programming. It offers the following patterns for increasing fault tolerance due to network problems or failure of any of the multiple services:

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



TYPICAL SCENARIO IN MICROSERVICES

eazy bytes

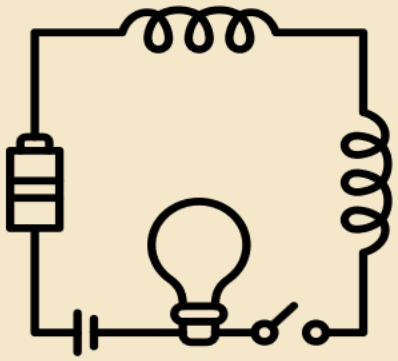


When a microservice responds slowly or fails to function, it can lead to the depletion of resource threads on the Edge server and intermediate services. This, in turn, has a negative impact on the overall performance of the microservice network.

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

CIRCUIT BREAKER PATTERN

eazy bytes



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

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

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

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

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

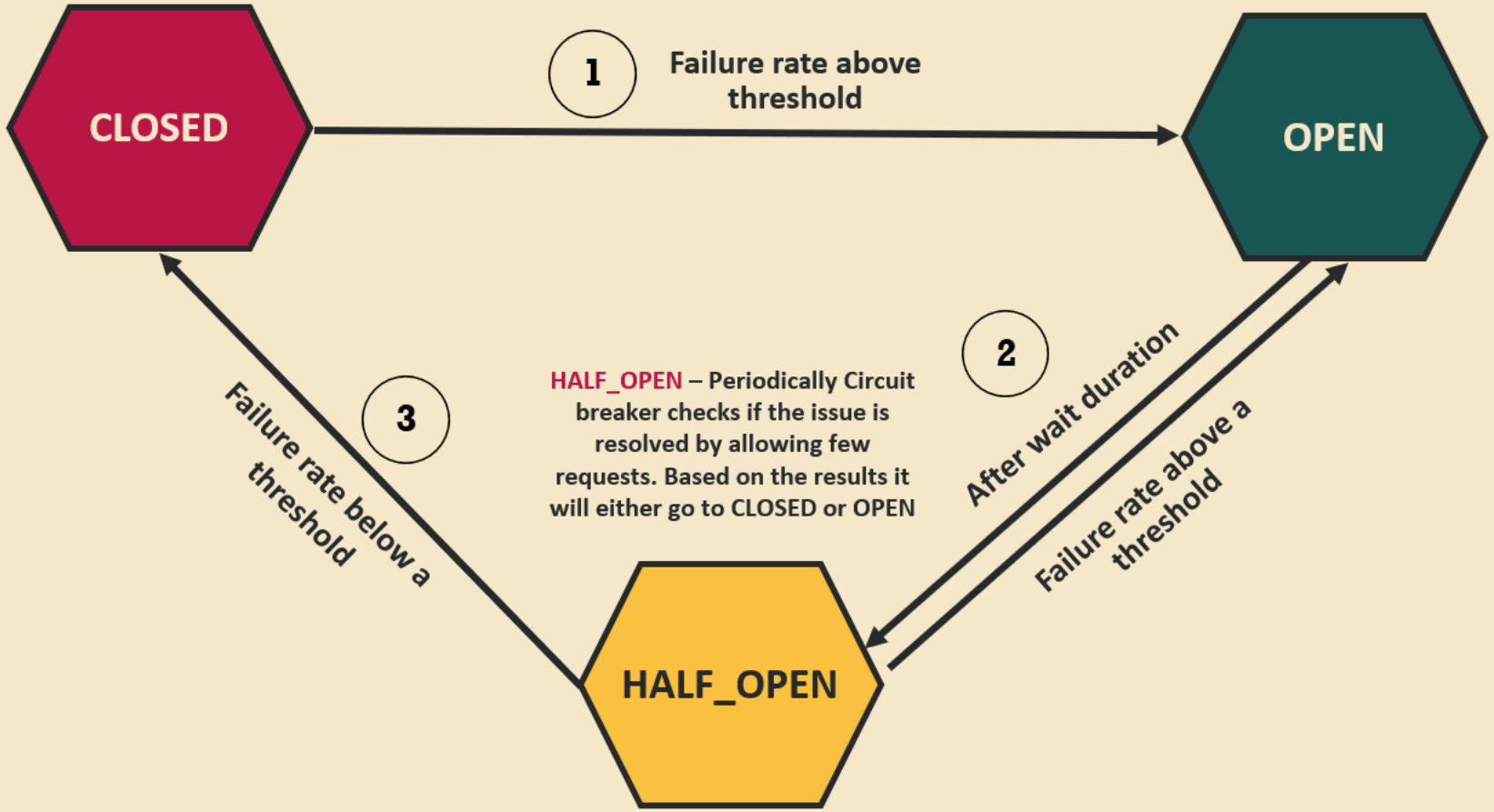
The advantages with circuit breaker pattern are,

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

CIRCUIT BREAKER PATTERN

eazy bytes

In Resilience4j, the circuit breaker is implemented via three states



CIRCUIT BREAKER PATTERN

eazy bytes

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

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

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

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

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

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

CIRCUIT BREAKER PATTERN

eazy bytes

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

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

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

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

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

}
```

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

CIRCUIT BREAKER PATTERN

eazy bytes

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

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

RETRY PATTERN

eazy bytes

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

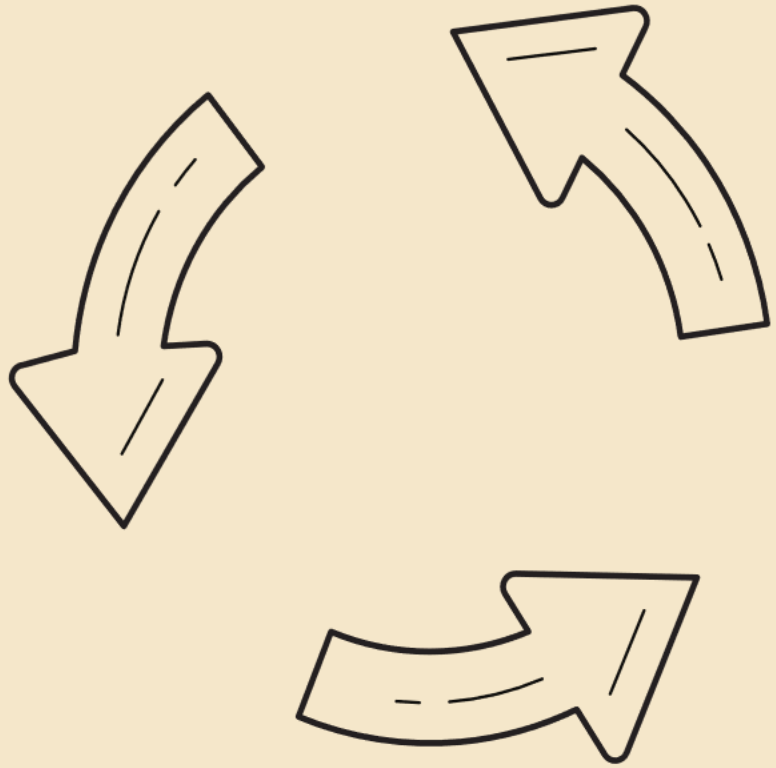
Here are some key components and considerations of implementing the Retry pattern in microservices:

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

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

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

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



RETRY PATTERN

eazy bytes

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

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

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

RETRY PATTERN

eazy bytes

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

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

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

}

private ResponseEntity<String> getBuildInfoFallBack(Throwable t) {

}
```

RETRY PATTERN

eazy bytes

2

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

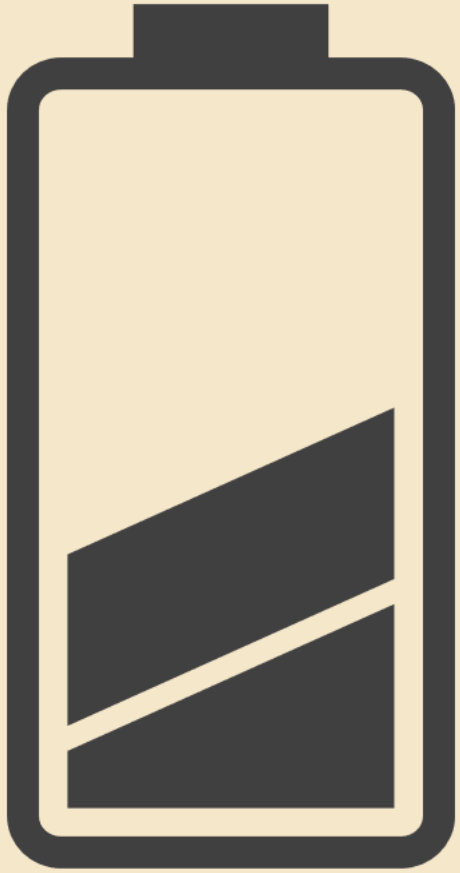
```
resilience4j.retry:
  configs:
    default:
      maxRetryAttempts: 3
      waitDuration: 500
      enableExponentialBackoff: true
      exponentialBackoffMultiplier: 2
      retryExceptions:
       - java.util.concurrent.TimeoutException
      ignoreExceptions:
       - java.lang.NullPointerException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

RATE LIMITTER PATTERN

eazy bytes

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

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



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

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

RATE LIMITTER PATTERN

eazy bytes

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

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

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

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

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

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

RATE LIMITTER PATTERN

eazy bytes

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

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

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

}

private ResponseEntity<String> getJavaVersionFallback(Throwable t) {

}
```

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

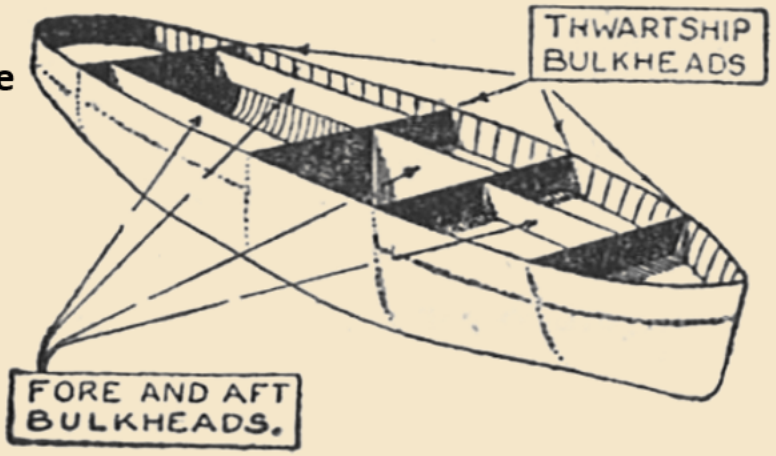
```
resilience4j.ratelimiter:
  configs:
    default:
      timeoutDuration: 5000
      limitRefreshPeriod: 5000
      limitForPeriod: 1
```

BULKHEAD PATTERN

eazy bytes

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

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

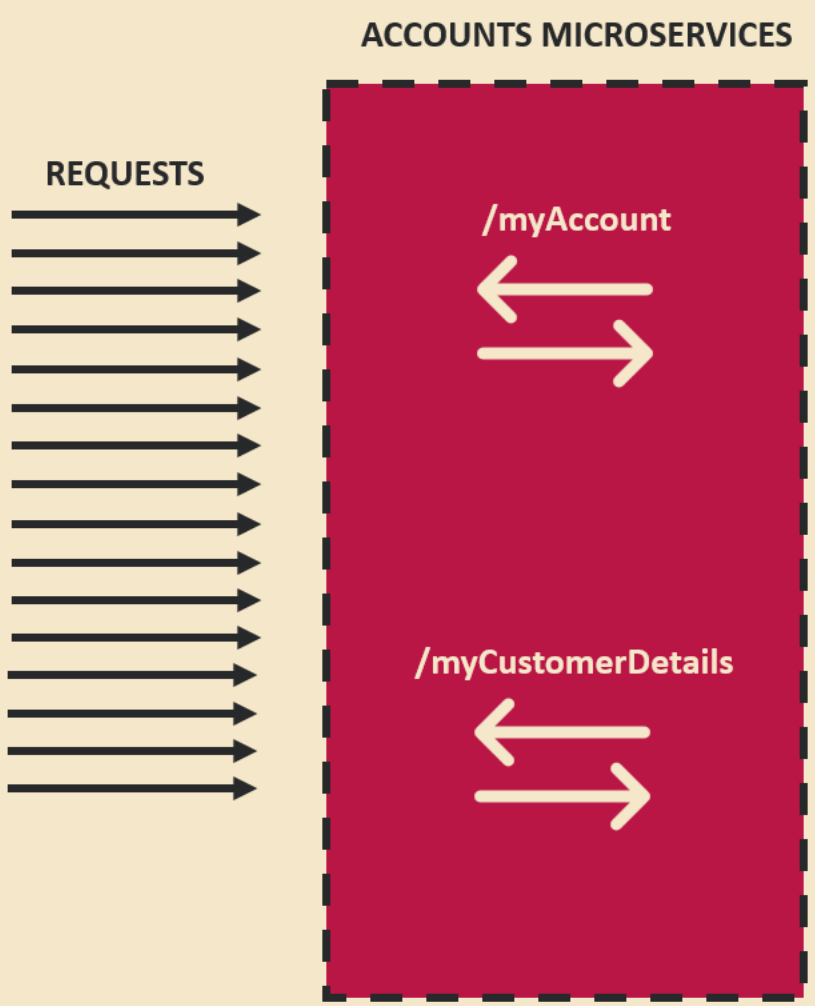


Bulkhead Pattern helps us to allocate limit the resources which can be used for specific services. So that resource exhaustion can be reduced.

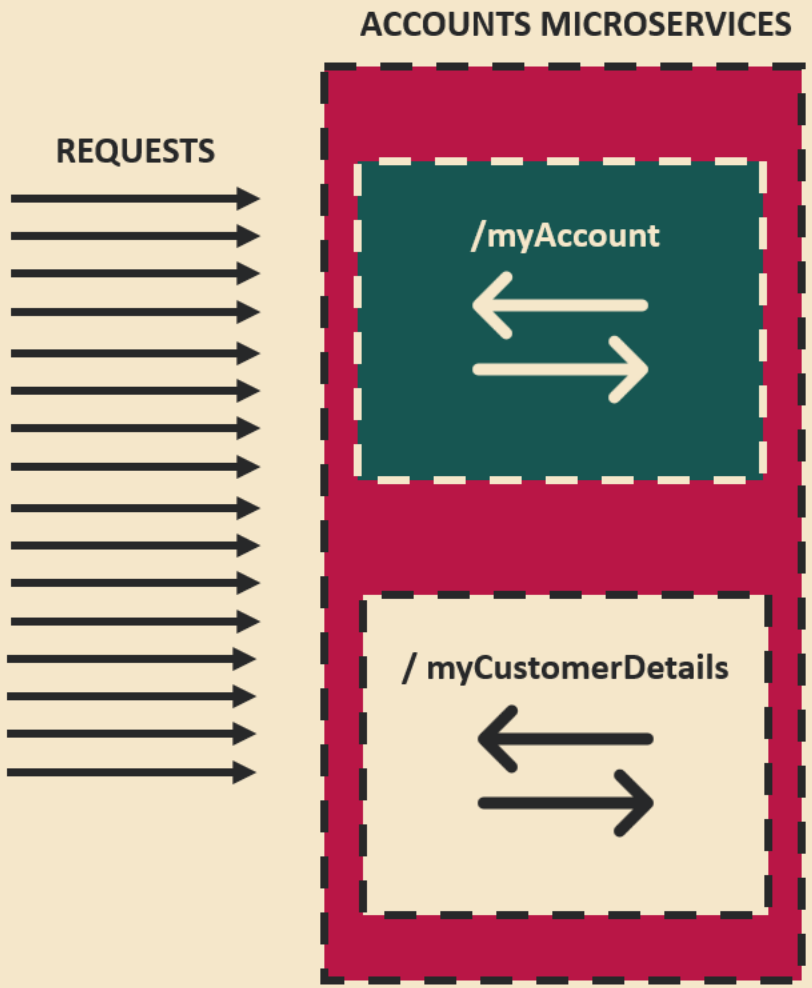
The Bulkhead pattern is particularly useful in systems that require high availability, fault tolerance, and isolation between components. By compartmentalizing components and enforcing resource boundaries, the Bulkhead pattern enhances the resilience and stability of the system, ensuring that failures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

BULKHEAD PATTERN

eazy bytes



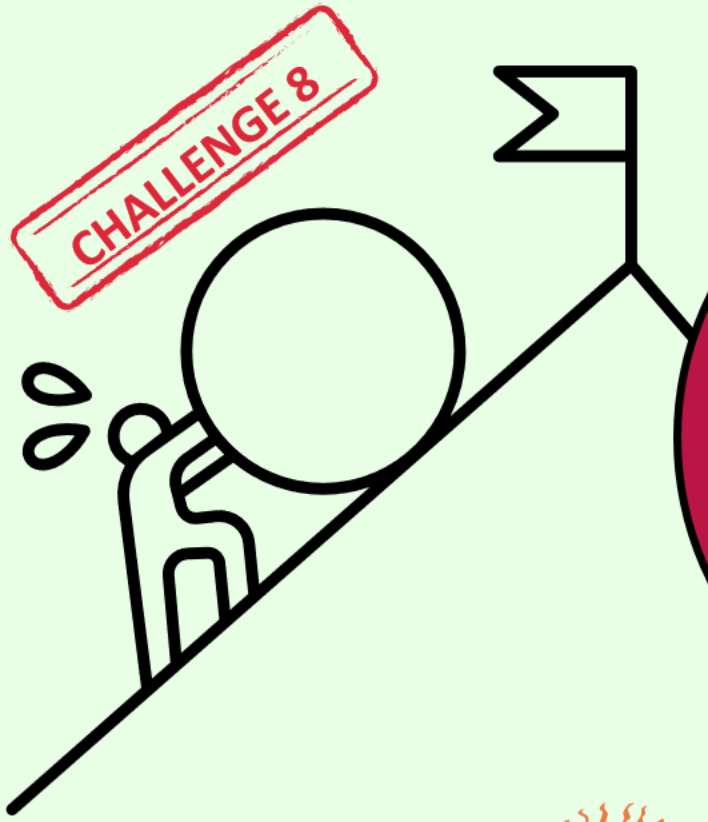
Without Bulkhead, /myCustomerDetails will start eating all the threads, resources available which will effect the performance of /myAccount



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

OBSERVABILITY AND MONITORING OF MICROSERVICES

eazy bytes



DEBUGGING A PROBLEM IN MICROSERVICES ?

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

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

MONITORING PERFORMANCE OF SERVICE CALLS?

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

MONITORING SERVICES METRICS & HEALTH ?

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

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

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

WHAT IS OBSERVABILITY ?

eazy bytes

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

The three pillars of observability are:

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

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

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

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

WHAT IS MONITORING ?

eazy bytes

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

Monitoring in microservices is important because it allows you to:

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

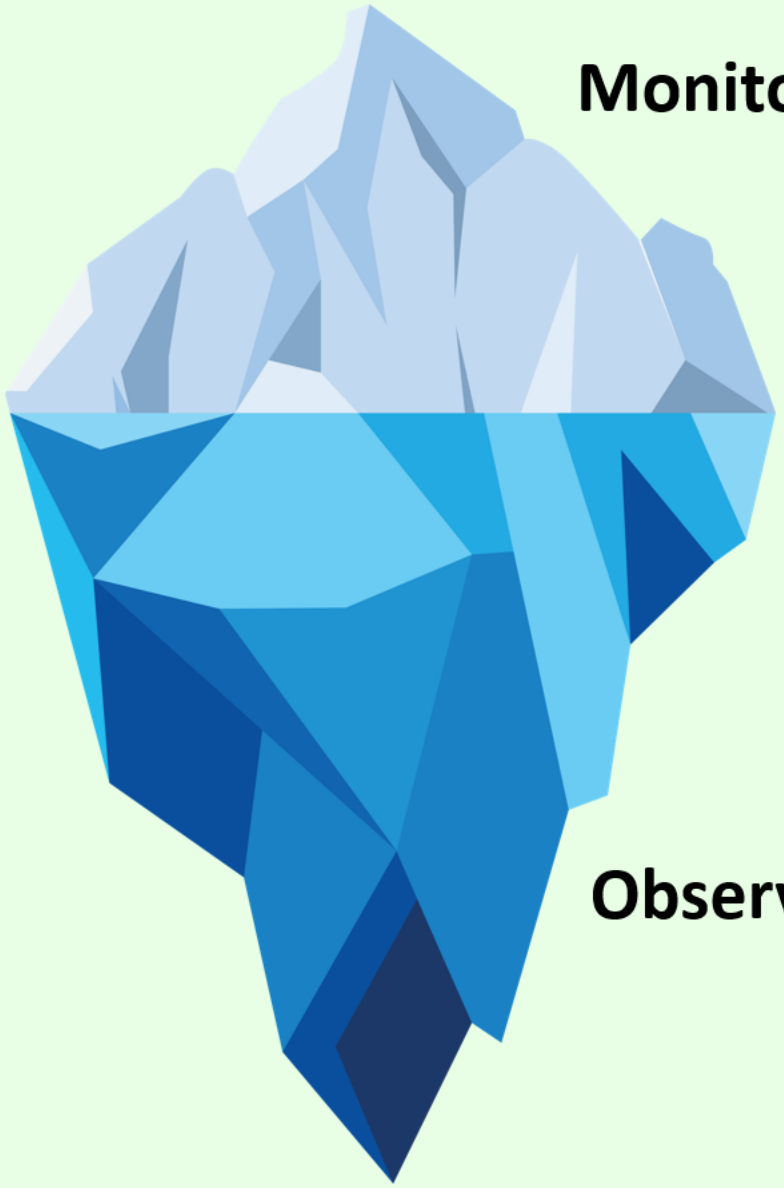
- Track the health of your microservices: **Monitoring can help you to track the health of your microservices, so you can identify any microservices that are underperforming or that are experiencing problems.**

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

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

Observability vs. Monitoring

eazy bytes



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

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

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

Logging

eazy bytes

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

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

Logging in Monolithic Apps

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

Logging in Microservices

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

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

Managing logs with Grafana, Loki & Promtail

eazy bytes

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

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

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

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

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

Grafana provides visualization of the log lines captured within Loki.

FROM GRAFANA LOKI 3.0,
Alloy replaced Promtail

Managing logs with Grafana, Loki & Alloy

eazy bytes

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

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

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

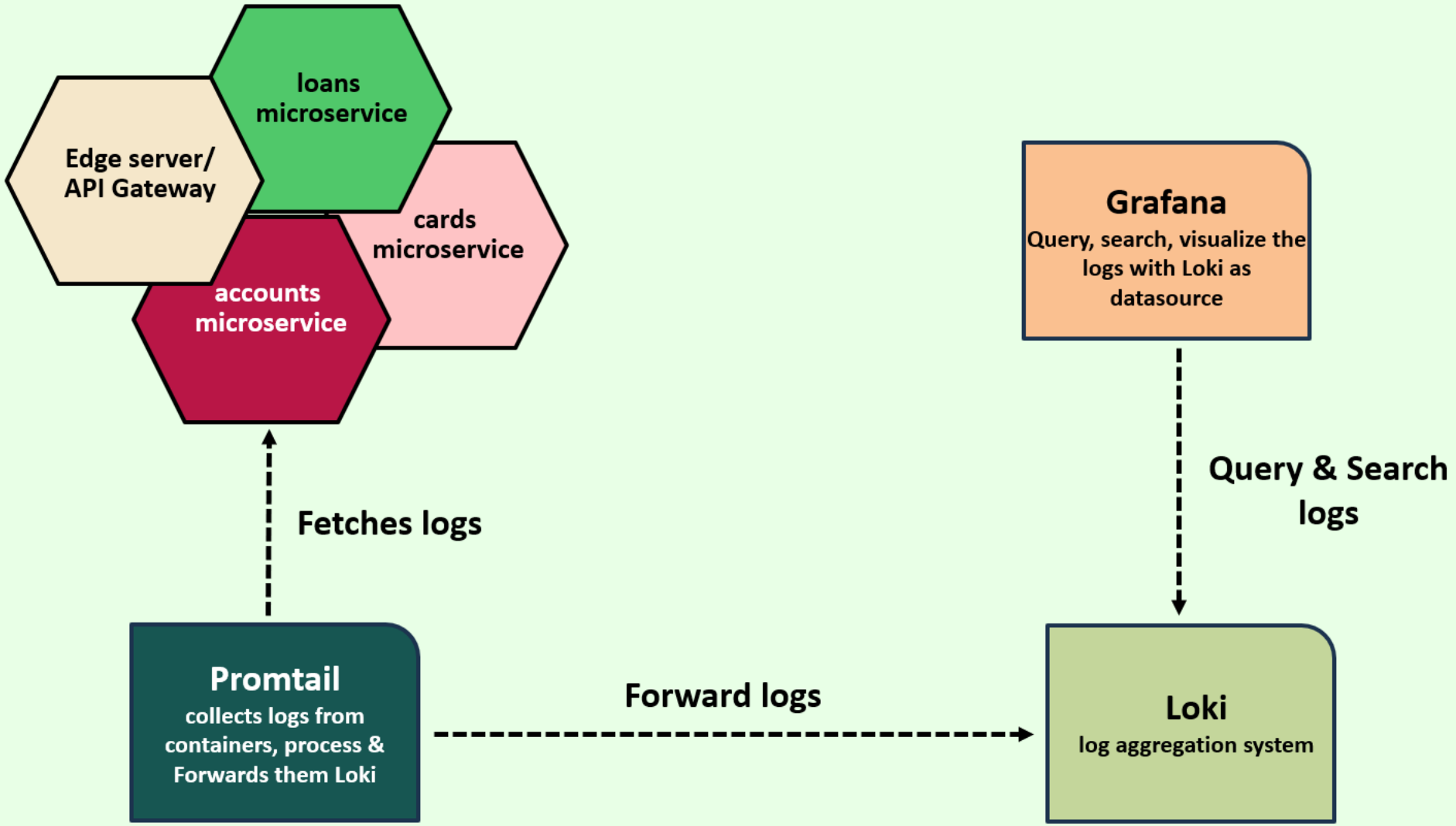
Grafana Alloy is a lightweight log agent that ships logs from your containers to Loki. It is easy to configure and can be used to collect logs from a wide variety of sources.

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

Grafana provides visualization of the log lines captured within Loki.

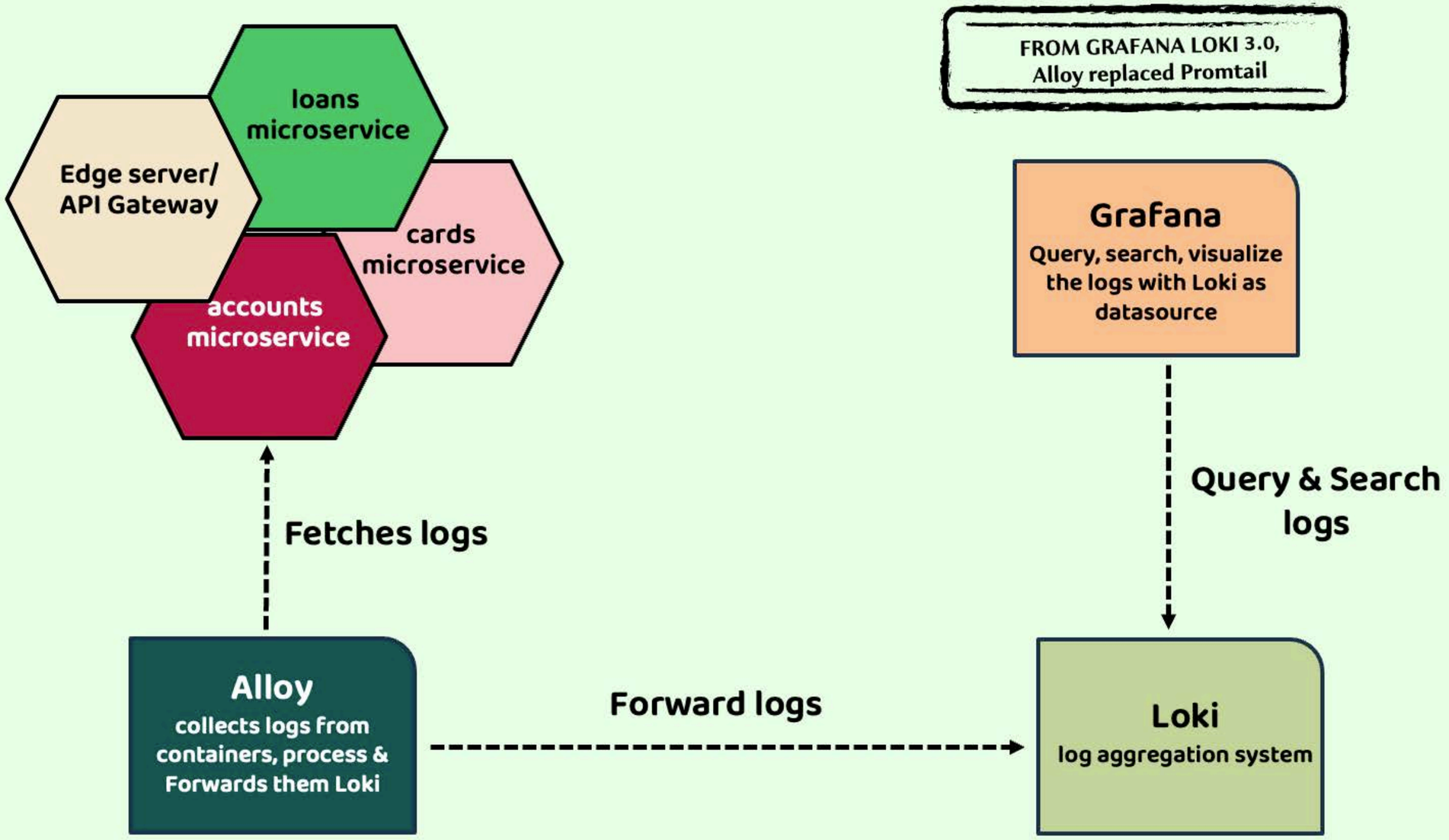
Managing logs with Grafana, Loki & Promtail

eazy bytes



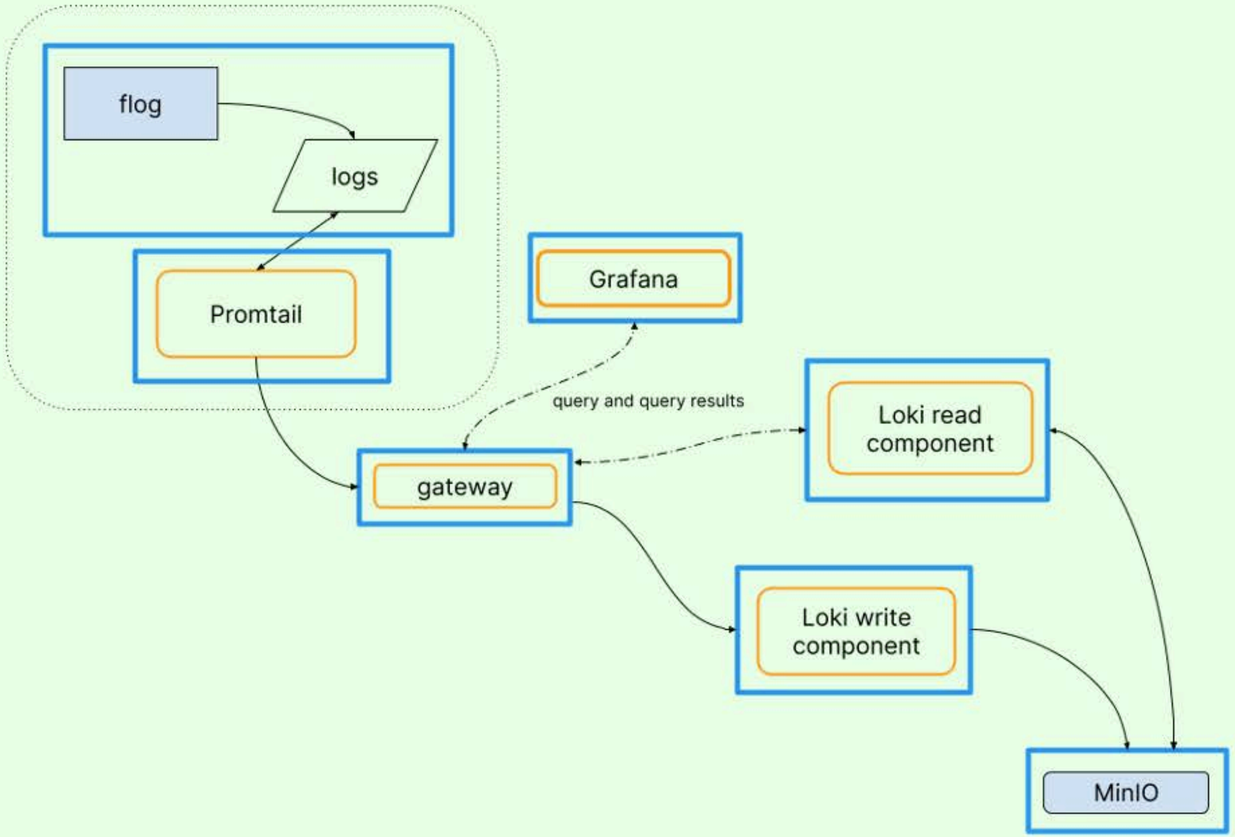
Managing logs with Grafana, Loki & Alloy

eazy bytes



Sample demo of logging using Grafana, Loki & promotail

eazy bytes



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

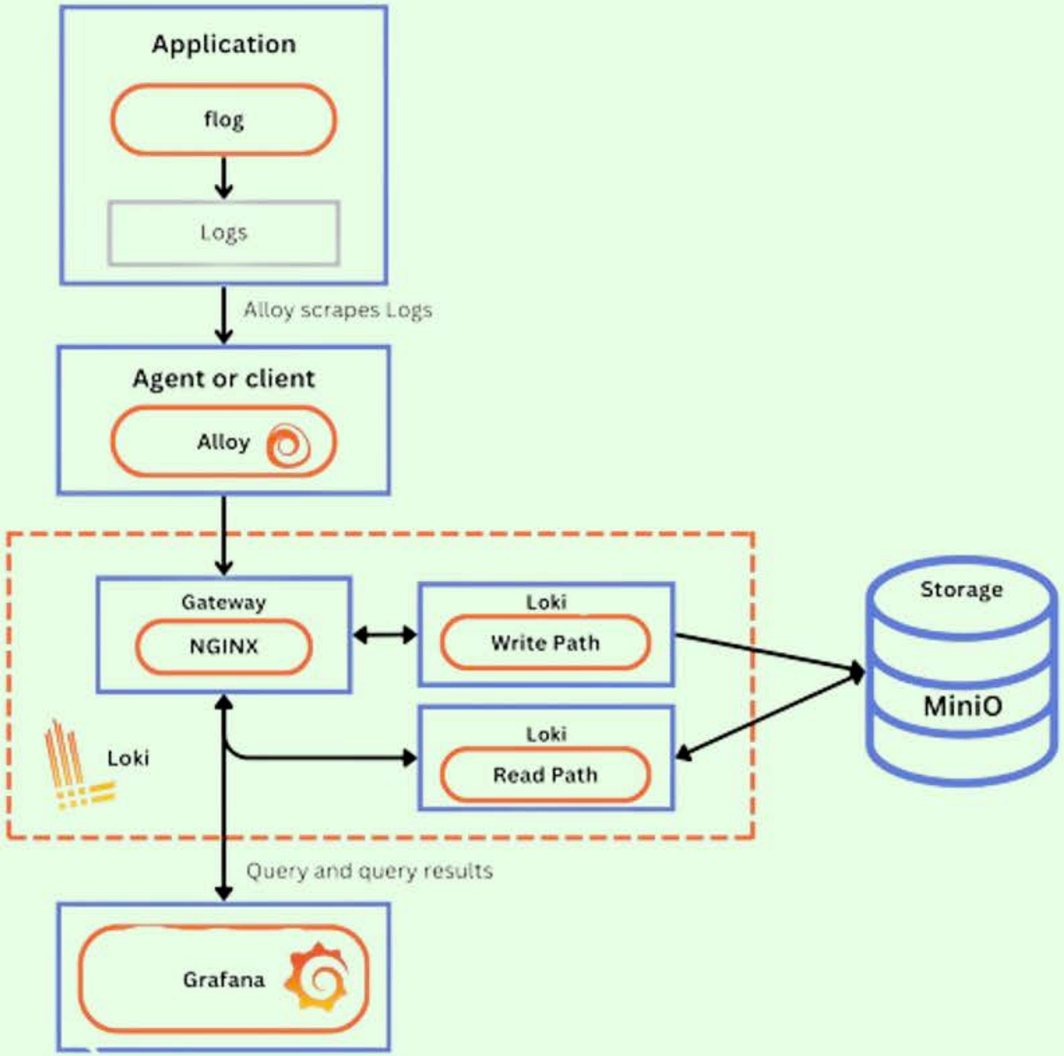
One advantage of treating logs as event streams and emitting them to stdout is that it decouples the application from the log processing infrastructure. The application can focus on its core functionality without being tied to a specific logging implementation or storage solution. The infrastructure,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

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

Sample demo of logging using Grafana, Loki & Alloy

eazy bytes

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



FROM GRAFANA LOKI 3.0,
Alloy replaced Promtail

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

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

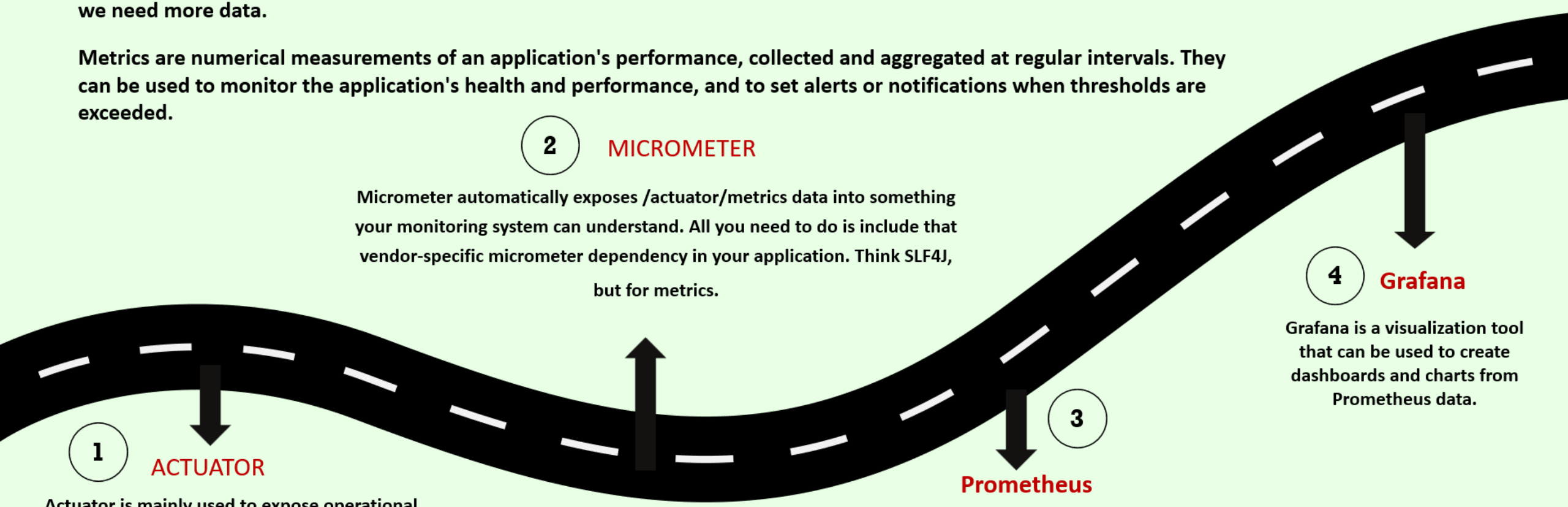
15-Factor methodology recommends the same to treat logs as events streamed to the standard output and not concern with how they are processed or stored.

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

eazy bytes

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

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



2 MICROMETER

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

4 Grafana

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

1 ACTUATOR

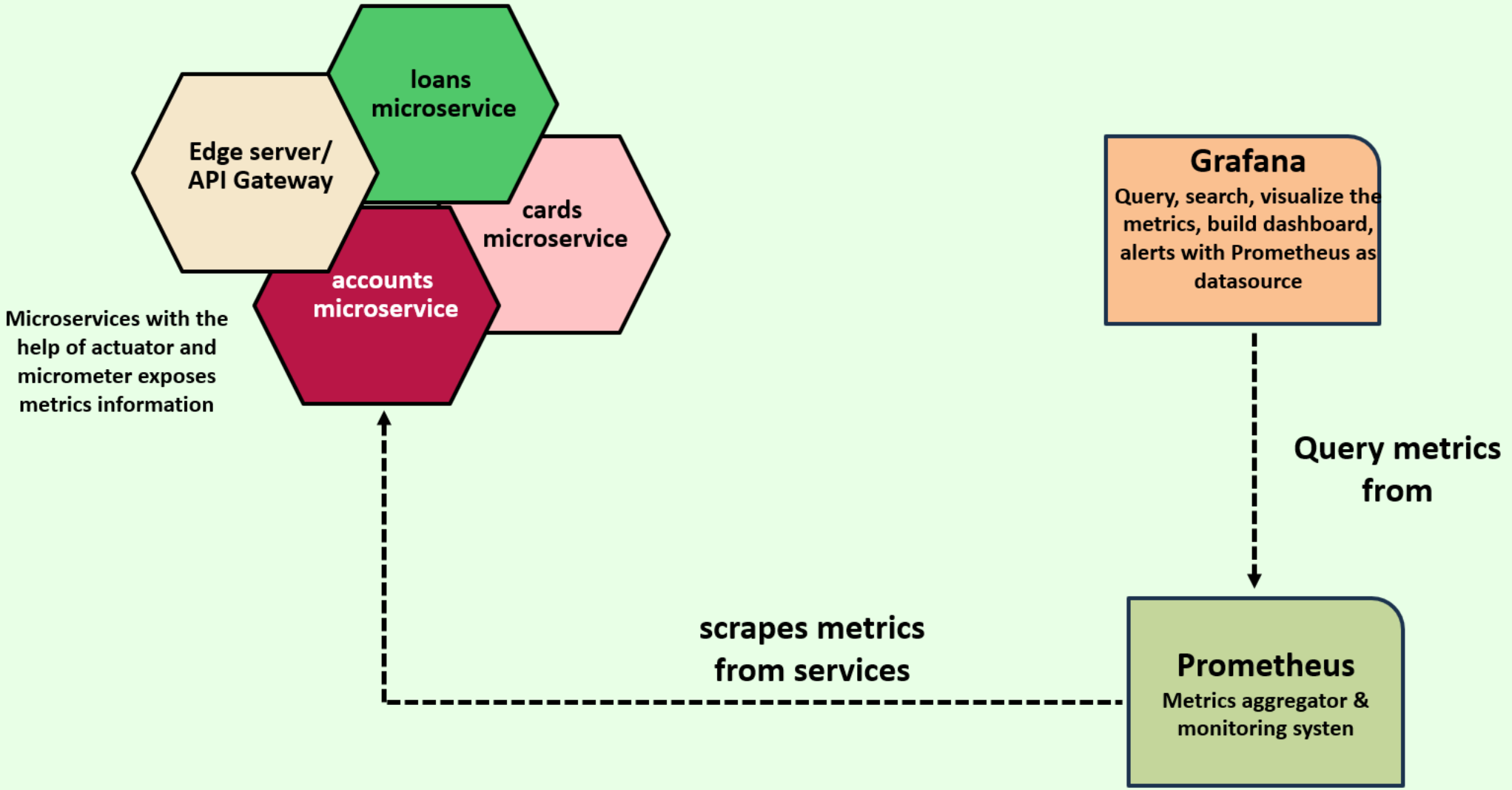
Actuator is mainly used to expose operational information about the running application — health, metrics, info, dump, env, etc. It uses HTTP endpoints or JMX beans to enable us to interact with it.

3 Prometheus

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

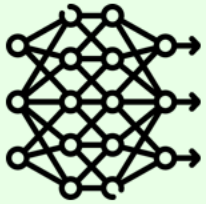
Metrics & monitoring with Spring Boot Actuator, Micrometer, Prometheus & Grafana

eazy bytes



Distributed tracing in microservices

eazy bytes



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

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

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

Distributed tracing encompasses three primary concepts:

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

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

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

Distributed tracing in microservices

eazy bytes

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

[gatewayserver, 29cdbe2e21bc, 259d5208]

Edge server/ API Gateway

accounts microservice

loans microservice

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

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

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

[accounts, 29cdbe2e21bc, 61fd3a92]

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

cards microservice

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

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

Distributed tracing with OpenTelemetry, Tempo & Grafana

eazy bytes

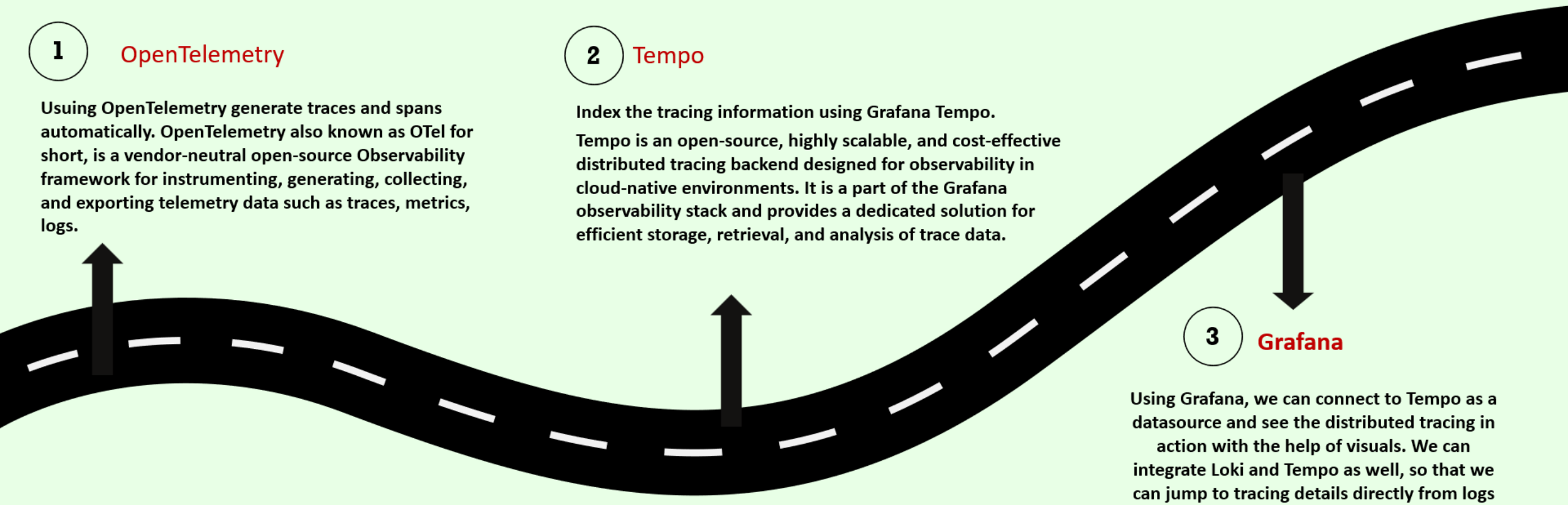
1 OpenTelemetry

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

2 Tempo

Index the tracing information using Grafana Tempo.

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

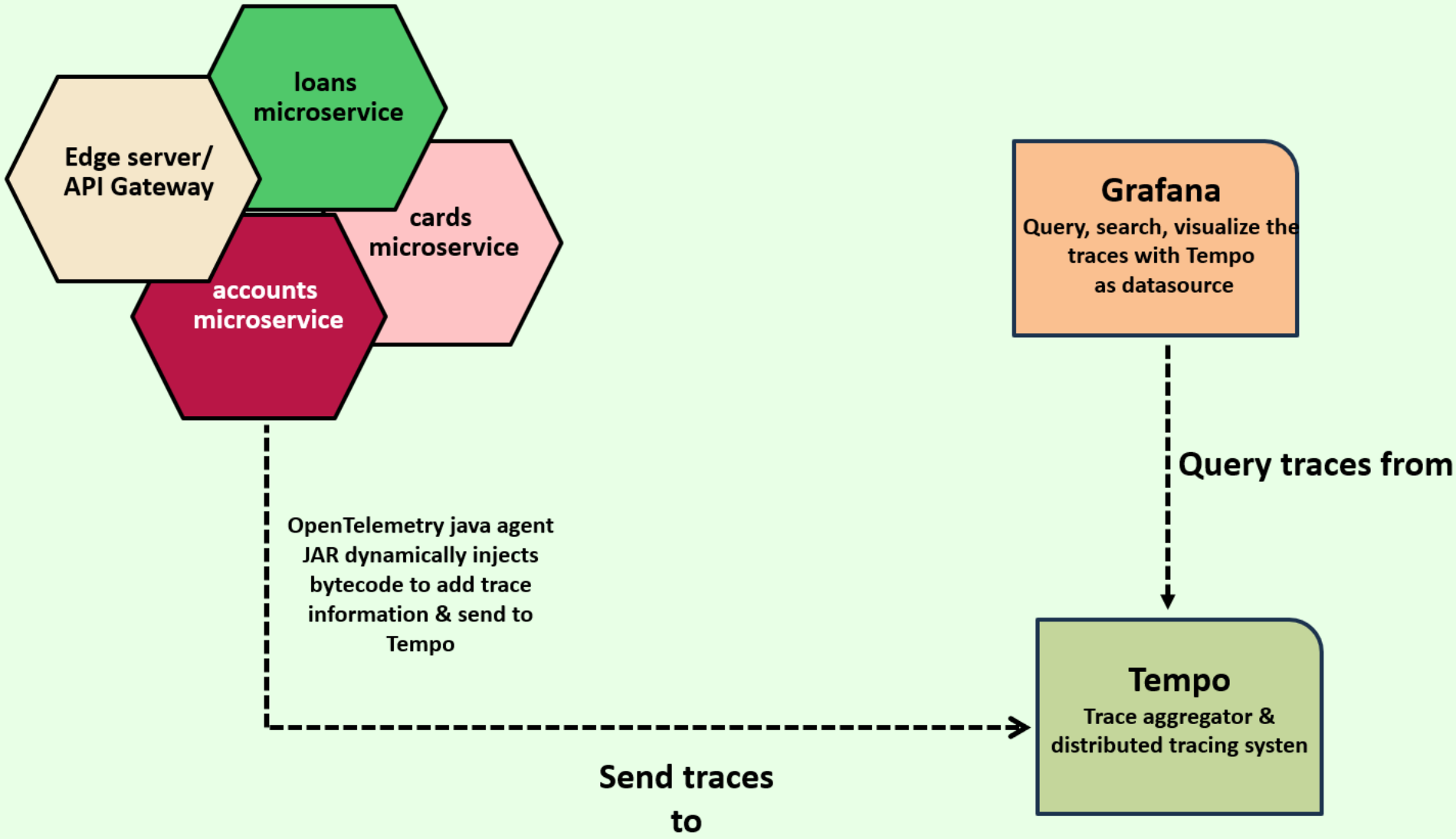


3 Grafana

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

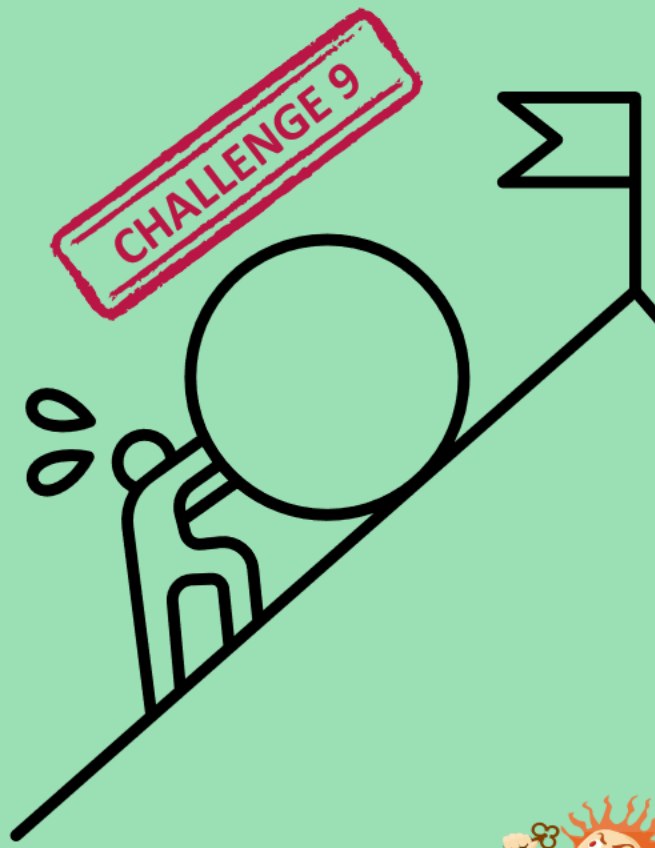
Distributed tracing with OpenTelemetry, Tempo & Grafana

eazy bytes



MICROSERVICES SECURITY

eazy bytes



SECURING MICROSERVICES FROM UNAUTHORIZED ACESS ?

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

AUTHENTICATION AND AUTHORIZATION

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

CENTRALIZED IDENTITY AND ACCESS MANAGEMENT (IAM)

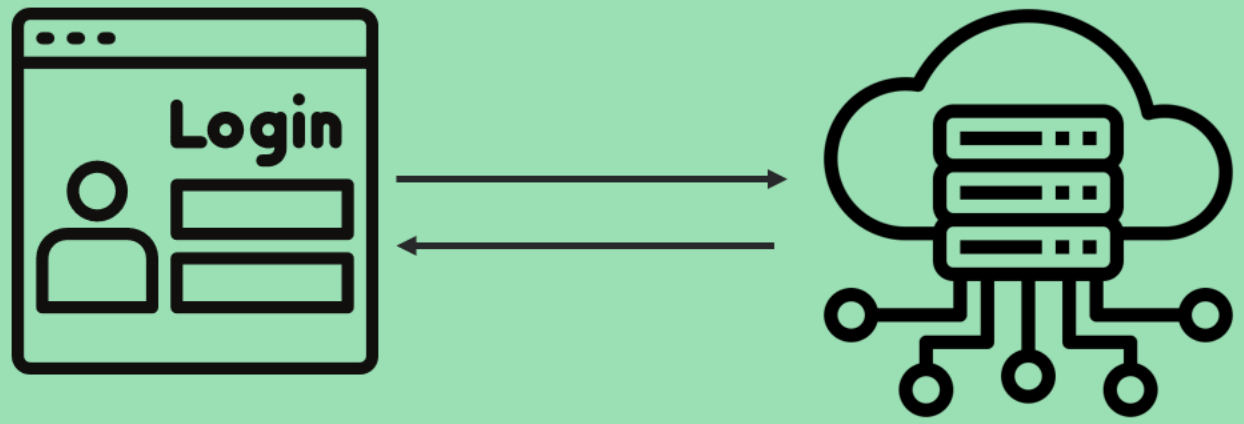
How to maintain a centralized component to store user credentials, handling identity & access management

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

PROBLEM THAT OAUTH2 SOLVES

eazy bytes

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



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

Drawbacks of Basic authentication

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

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

PROBLEM THAT OAUTH2 SOLVES

eazy bytes

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

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

INTRODUCTION TO OAUTH2

eazy bytes

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

Below are the few advantages of OAuth2 standard,

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

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

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

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

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

OAUTH2 TERMINOLOGY

eazy bytes

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

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

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

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

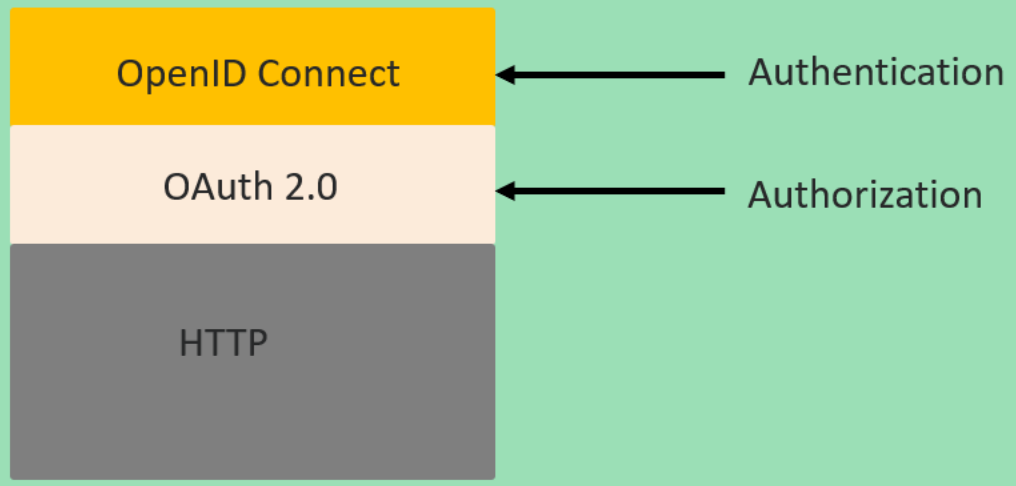
Scopes – These are the granular permissions the Client wants, such as access to data or to perform certain actions. The Auth server can issue an access token to client with the scope of Email, READ etc.

WHAT IS OPENID CONNECT & WHY IT IS IMPORTANT ?

eazy bytes

What is OpenID Connect?

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



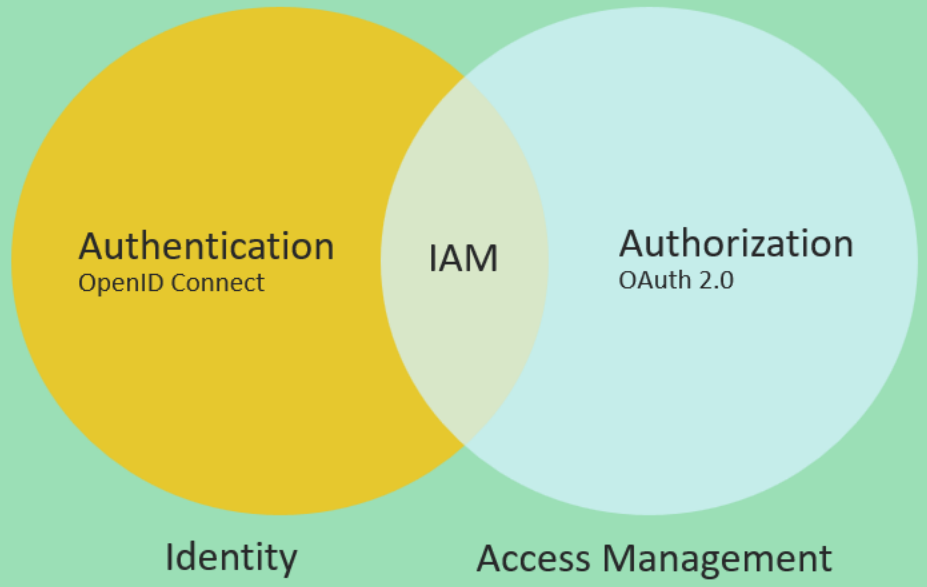
The OpenID Connect flow looks the same as OAuth. The only differences are, in the initial request, a specific scope of openid is used, and in the final exchange the client receives both an Access Token and an ID Token.

WHAT IS OPENID CONNECT & WHY IT IS IMPORTANT ?

eazy bytes

Why is OpenID Connect important?

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

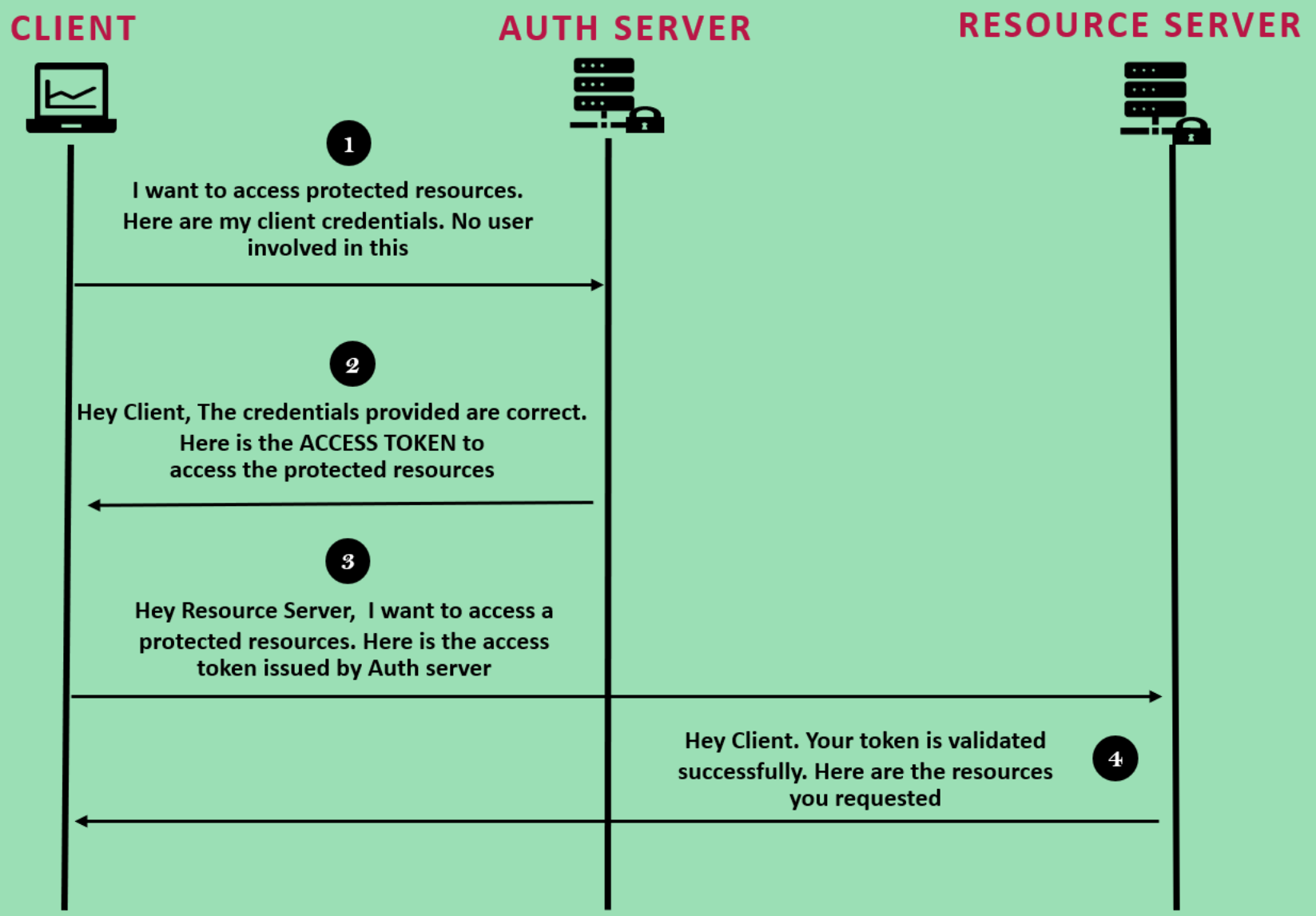


OpenID Connect add below details to OAuth 2.0

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

CLIENT CREDENTIALS GRANT TYPE FLOW IN OAUTH2

eazy bytes



CLIENT CREDENTIALS GRANT TYPE FLOW IN OAUTH2

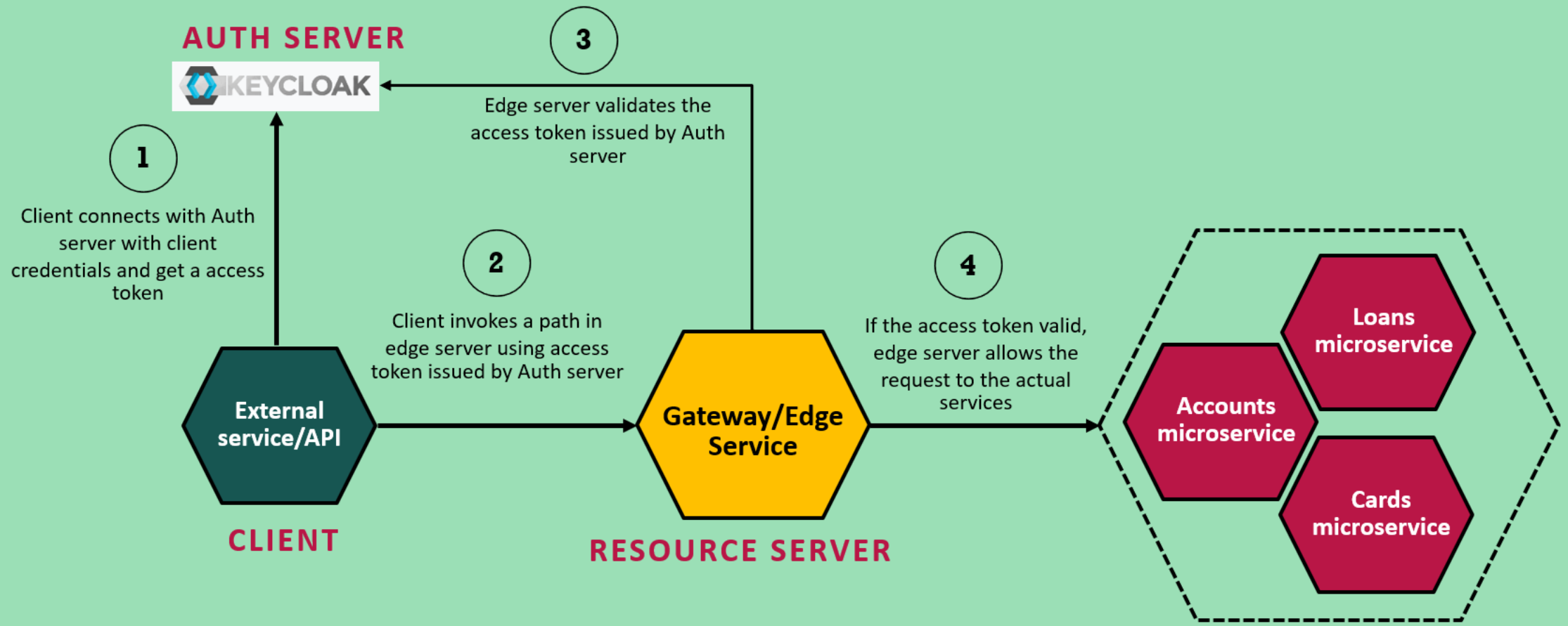
eazy bytes

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

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

SECURING GATEWAY USING CLIENT CREDENTIALS GRANT TYPE FLOW IN OAUTH2

eazy bytes

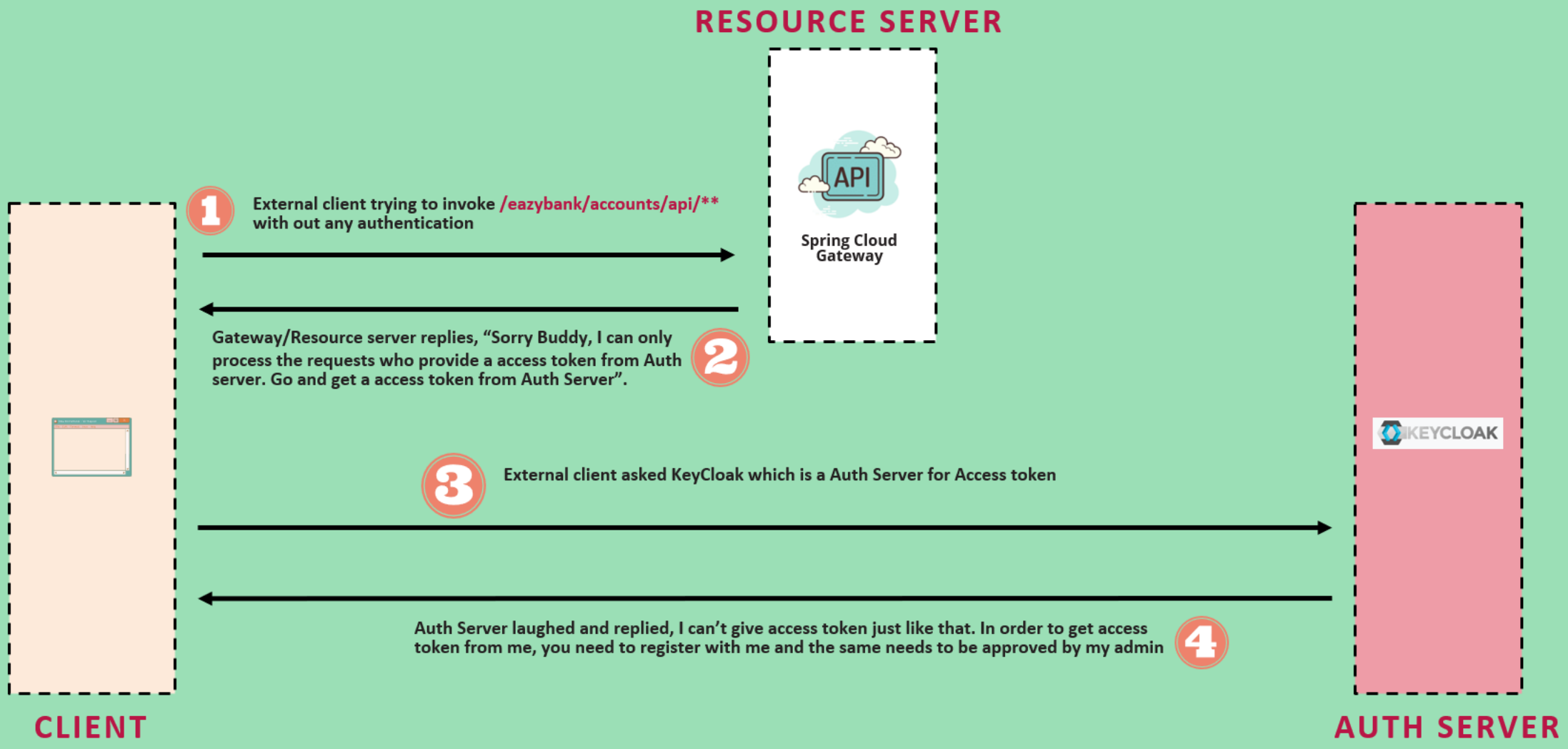


Unsecured services deployed behind the docker network or Kubernetes firewall network. So can't be accessed directly

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

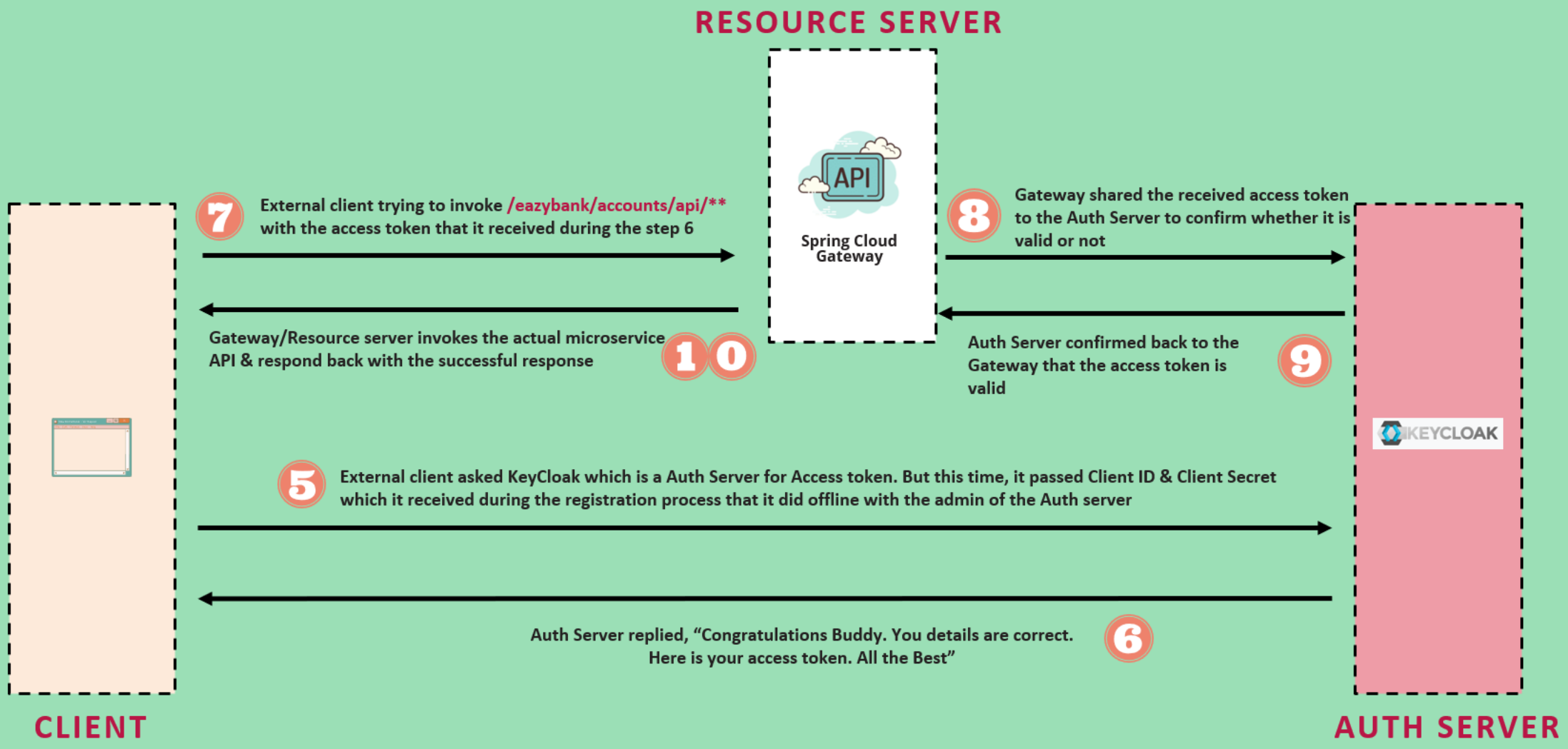
SECURING GATEWAY USING CLIENT CREDENTIALS GRANT TYPE FLOW IN OAUTH2

eazy bytes



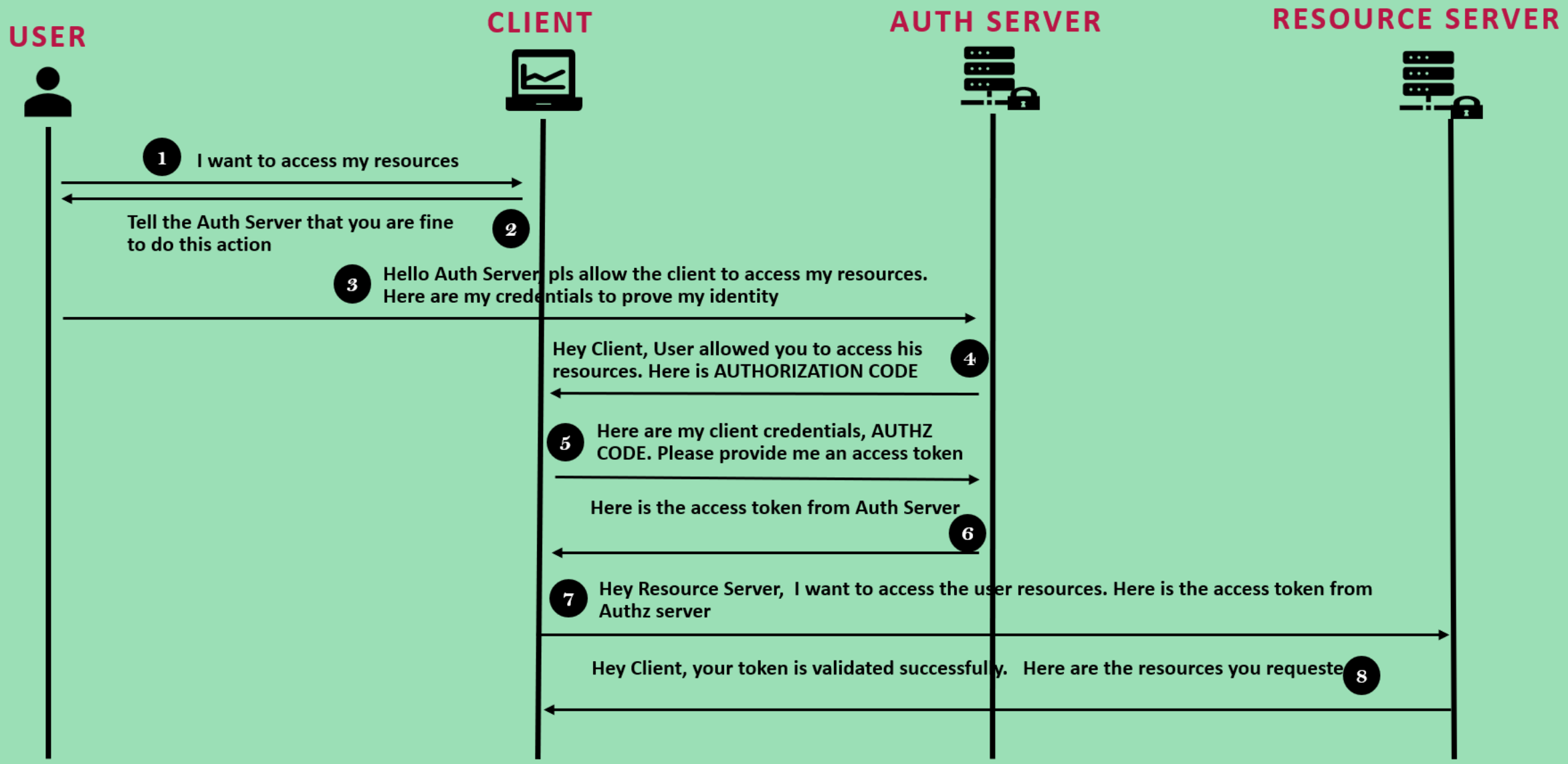
SECURING GATEWAY USING CLIENT CREDENTIALS GRANT TYPE FLOW IN OAUTH2

eazy bytes



AUTHORIZATION CODE GRANT TYPE FLOW IN OAUTH2

eazy bytes



AUTHORIZATION CODE GRANT TYPE FLOW IN OAUTH2

eazy bytes

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

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

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

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

AUTHORIZATION CODE GRANT TYPE FLOW IN OAUTH2

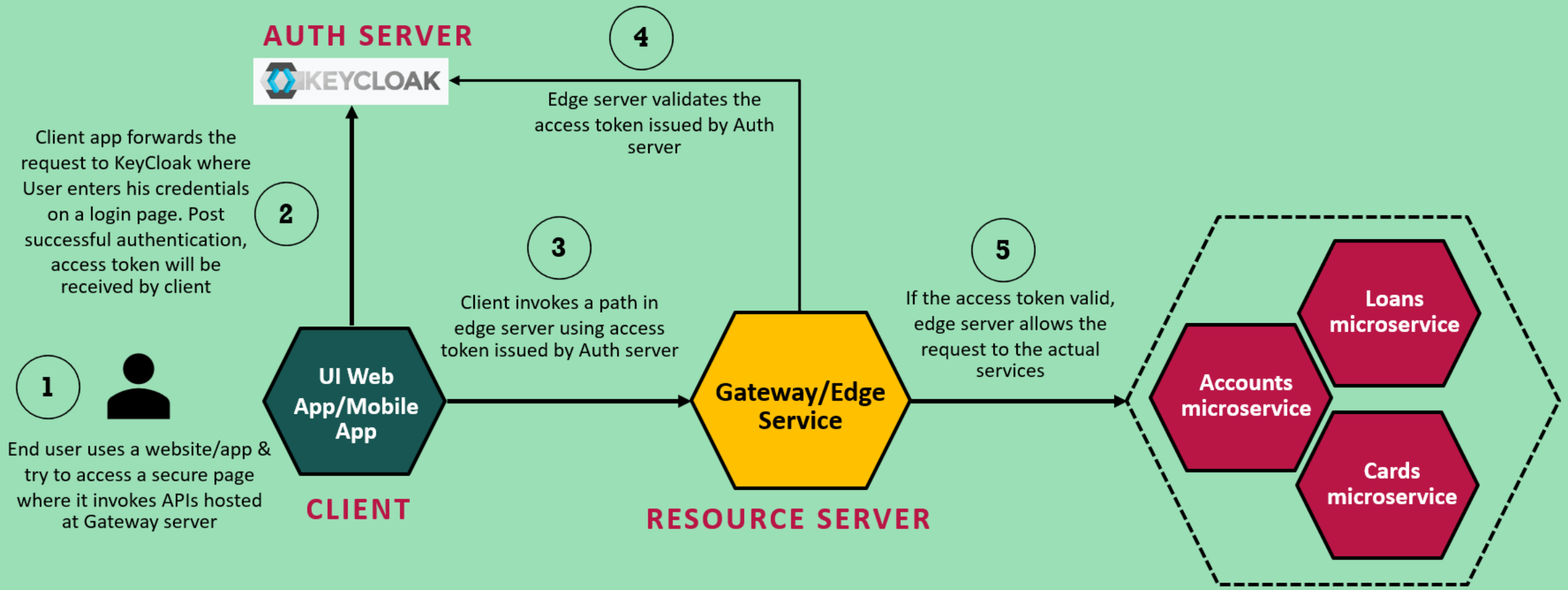
eazy bytes

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

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

SECURING GATEWAY USING AUTHORIZATION CODE GRANT TYPE FLOW IN OAUTH2

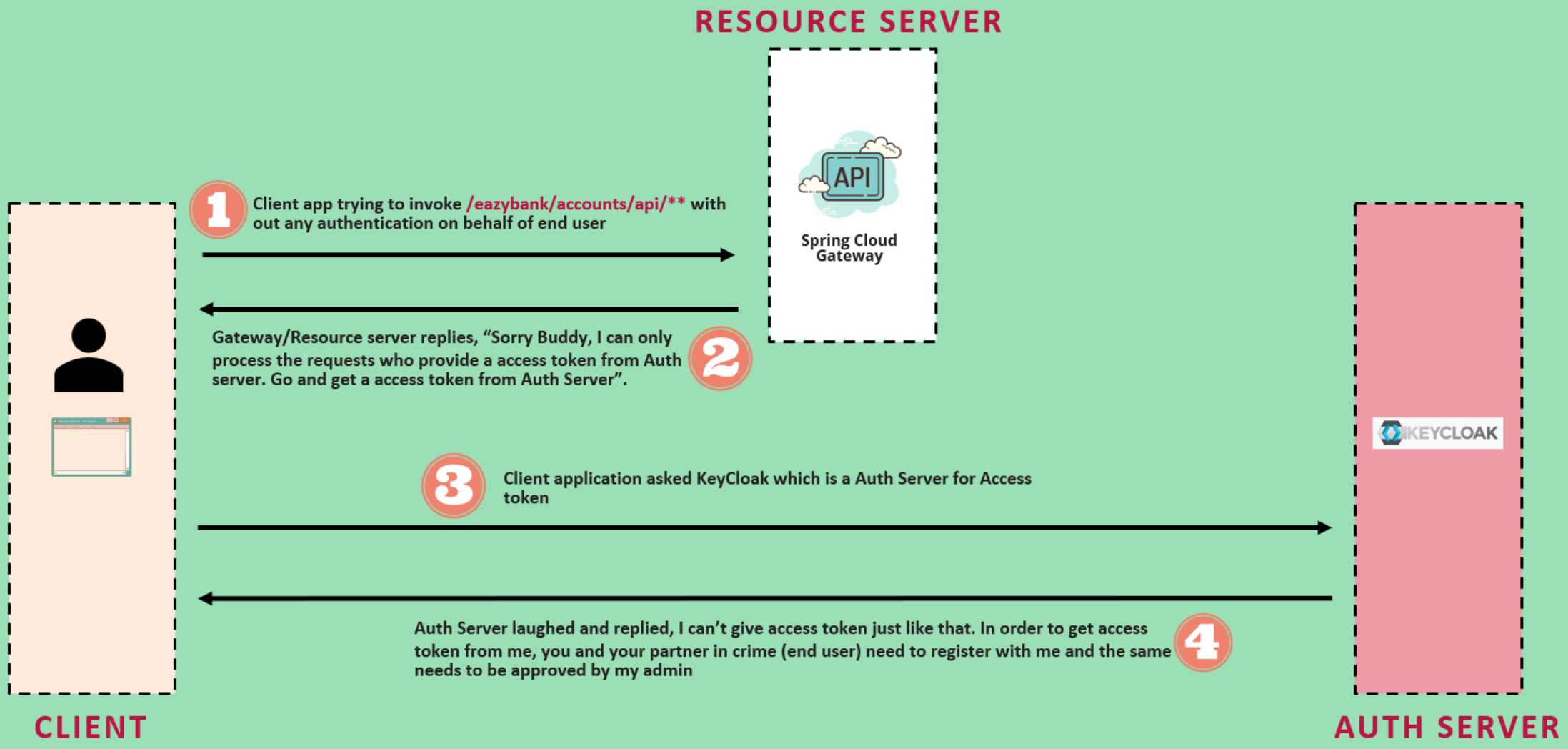
eazy bytes



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

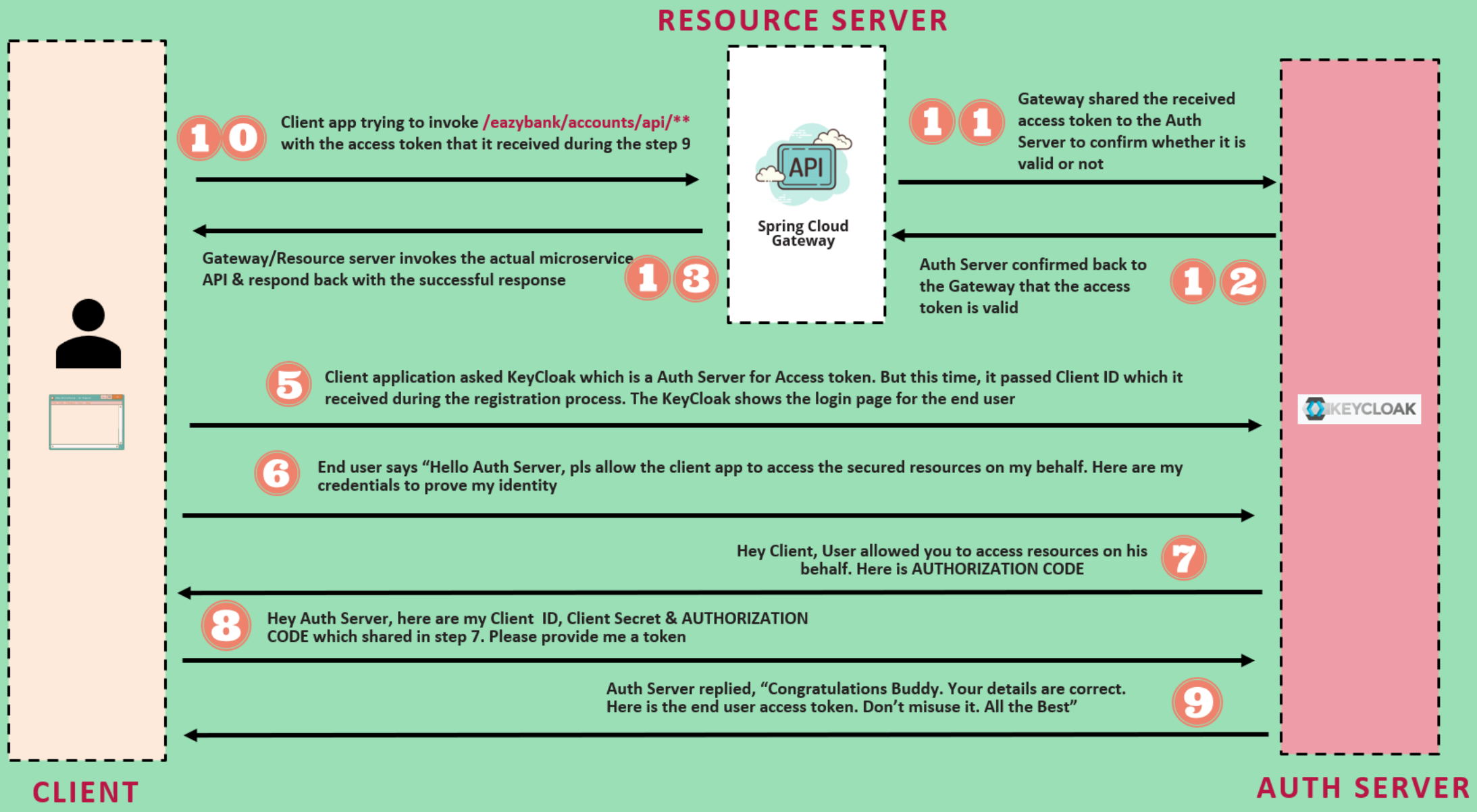
SECURING GATEWAY USING AUTHORIZATION CODE GRANT TYPE FLOW IN OAUTH2

eazy bytes



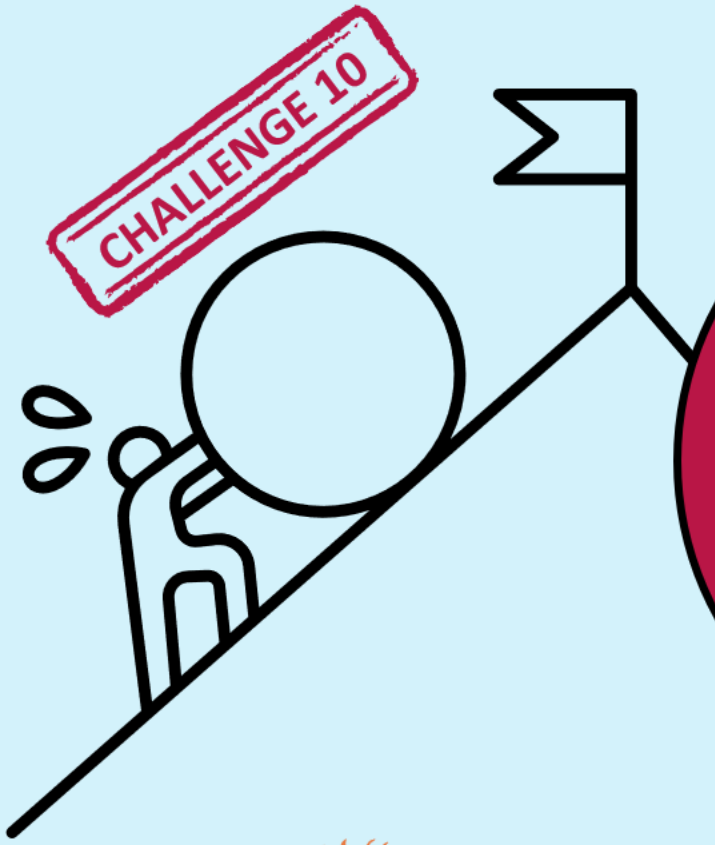
SECURING GATEWAY USING AUTHORIZATION CODE GRANT TYPE FLOW IN OAUTH2

eazy bytes



Building Event-driven microservices

eazy bytes



Avoiding temporal coupling whenever possible

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

Using asynchronous communication

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

Building event driven microservices

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

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

Event-driven models

eazy bytes

Event-driven architectures can be built using two primary models

Publisher/Subscriber (Pub/Sub) Model

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

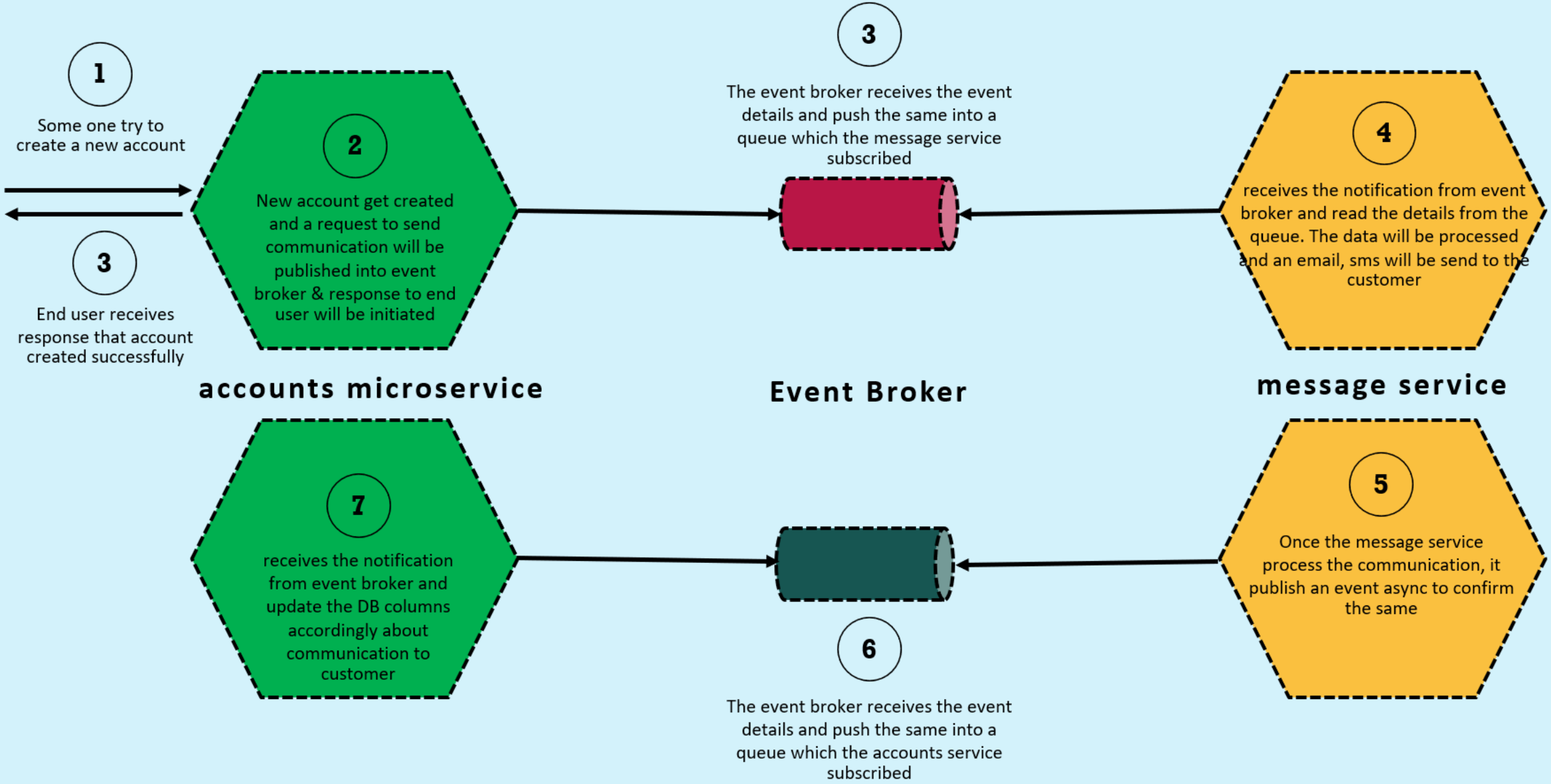
Event Streaming Model

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

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

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

eazy bytes



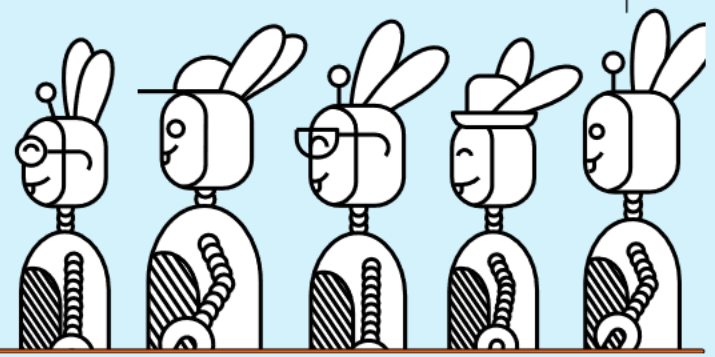
Using RabbitMQ for publish/subscribe communications

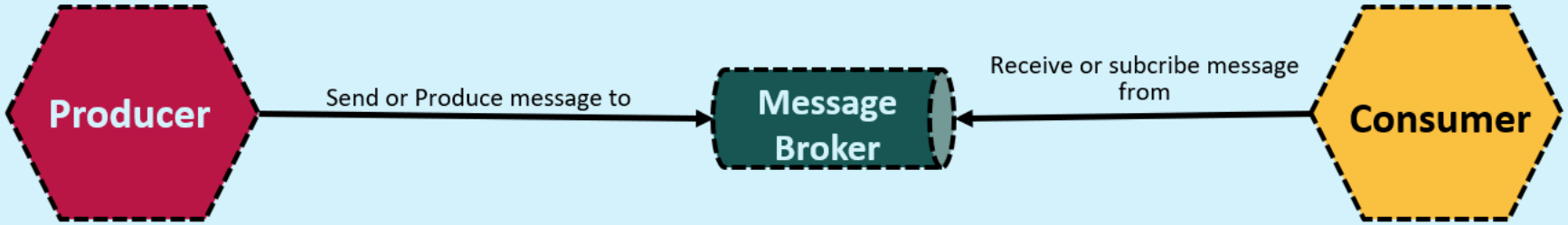
eazy bytes

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

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

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

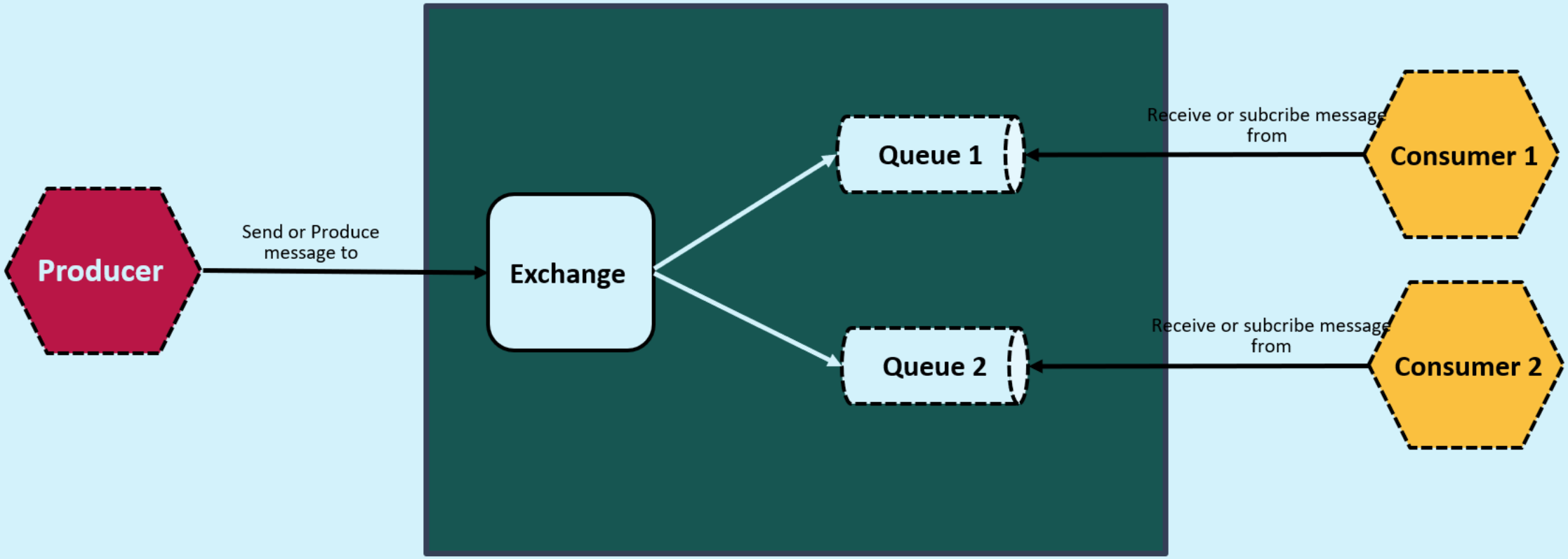




Using RabbitMQ for publish/subscribe communications

eazy bytes

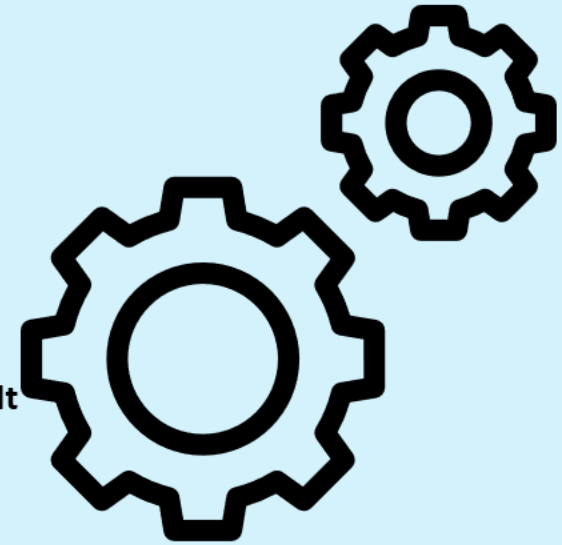
The messaging model of AMQP operates on the principles of **exchanges** and **queues**, as depicted in the following illustration. Producers transmit messages to an exchange. Based on a specified routing rule, RabbitMQ determines the queues that should receive a copy of the message. Consumers, in turn,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



Why to use Spring Cloud Function ?

eazy bytes

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



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

Spring Cloud Function features:

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

Steps to create functions using Spring Cloud Functions

eazy bytes

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

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

2 **Implement the business logic using functions**

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

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

```java
@Configuration
public class MessageFunctions {

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

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

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

Steps to create functions using Spring Cloud Functions

eazy bytes

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

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

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

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

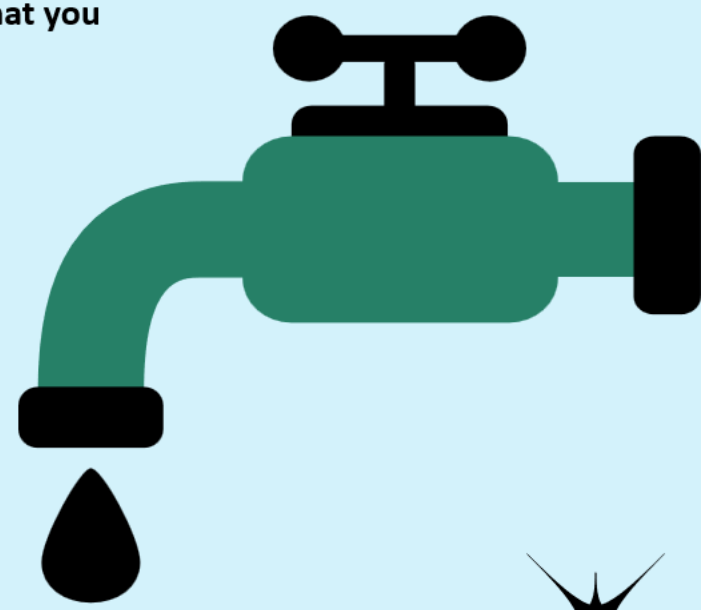
Based on the provided functions, the framework offers various ways to expose them according to our needs. For instance, Spring Cloud Function can automatically expose the functions specified in spring.cloud.function.definition as REST endpoints. This allows you to package the application, deploy it on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

Why to use Spring Cloud Stream ?

eazy bytes

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

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



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

The core building blocks of Spring Cloud Stream are:

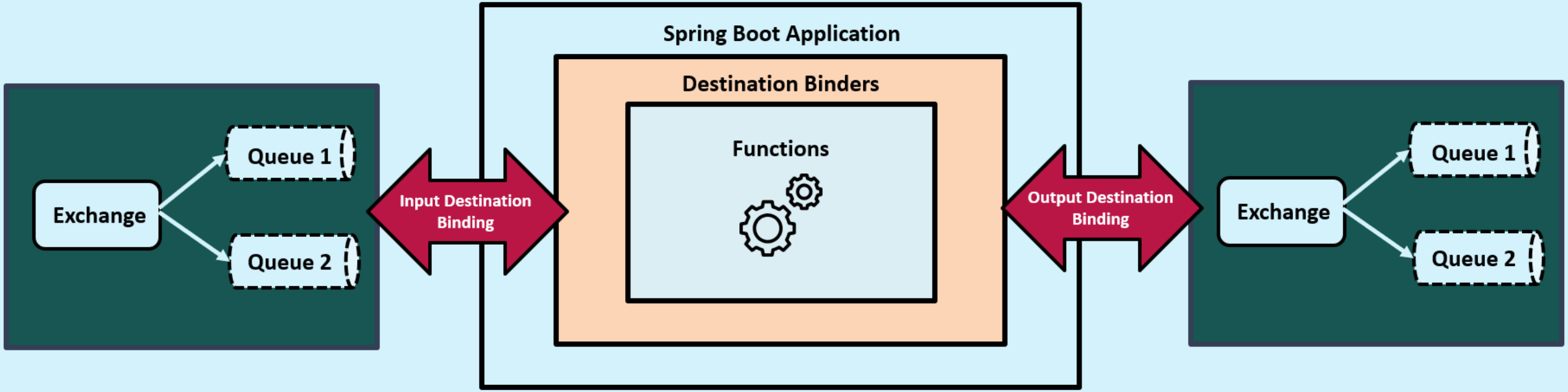
- **Destination Binders:** Components responsible to provide integration with the external messaging systems.
- **Destination Bindings:** Bridge between the external messaging systems and application code (producer/consumer) provided by the end user.
- **Message:** The canonical data structure used by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

Why to use Spring Cloud Stream ?

eazy bytes

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

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



Steps to create bindings using Spring Cloud Stream

eazy bytes

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

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

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

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

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

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

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

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

Event producing and consuming in accounts microservice

eazy bytes

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

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

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

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

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

Event producing and consuming in accounts microservice

eazy bytes

3

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

```
@Configuration
public class AccountsFunctions {

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

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

Event producing and consuming in accounts microservice

eazy bytes

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

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

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

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

Apache Kafka Vs RabbitMQ

eazy bytes

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

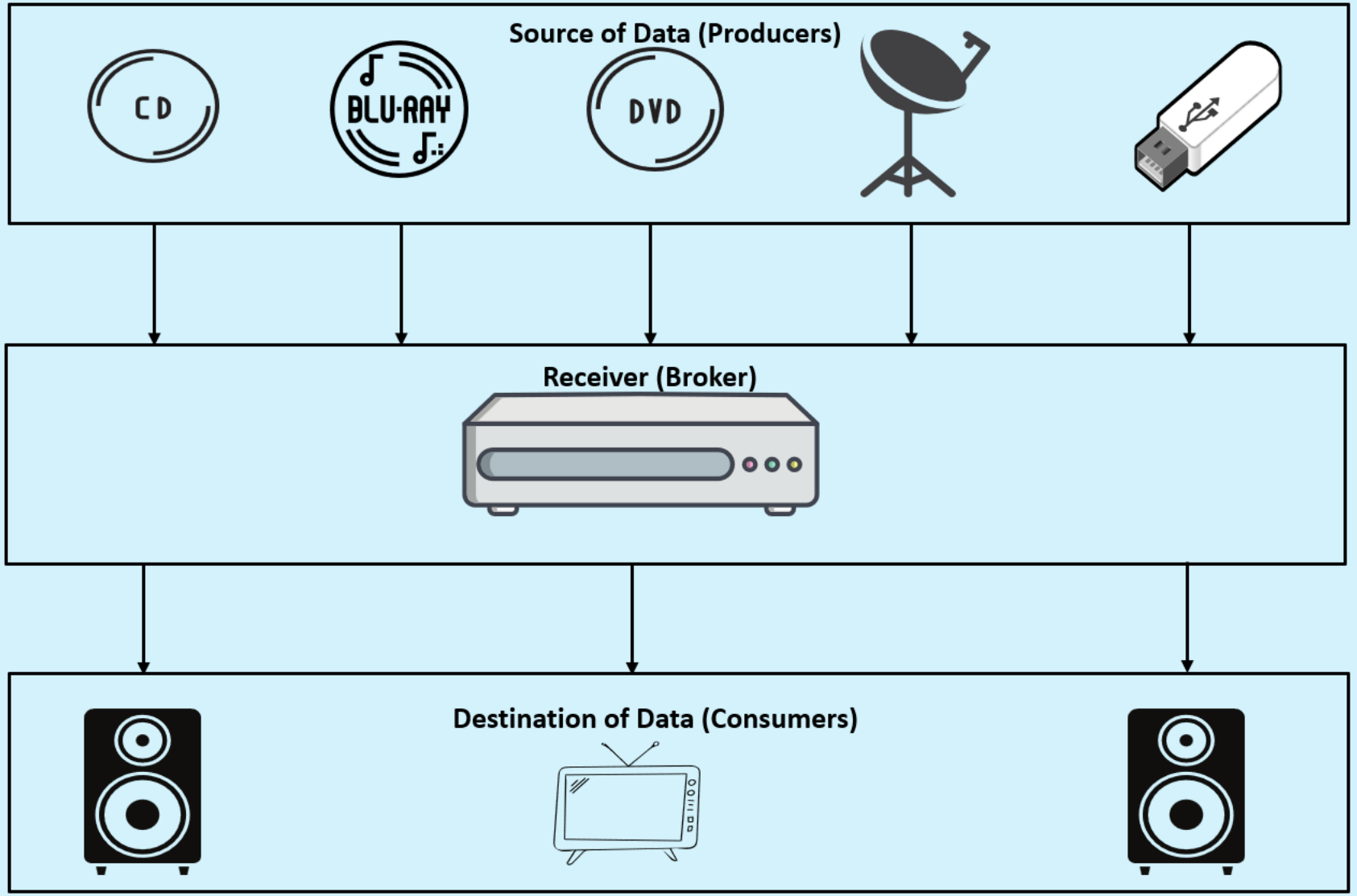
- **Design:** Kafka is a distributed event streaming platform, while RabbitMQ is a message broker. This means that Kafka is designed to handle large volumes of data, while RabbitMQ is designed to handle smaller volumes of data with more complex routing requirements.

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

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

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

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

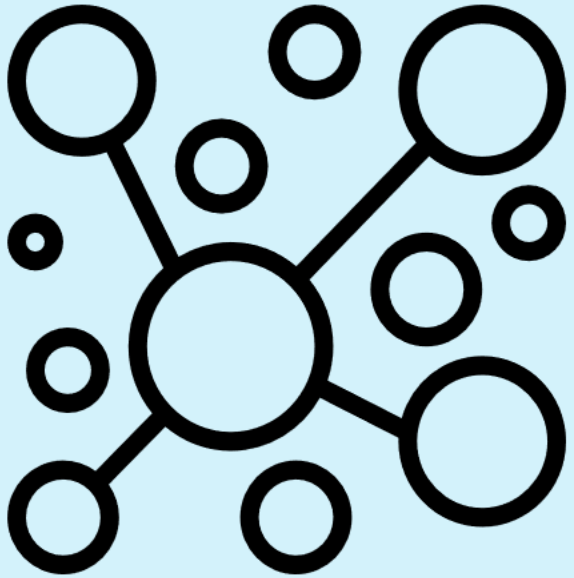

Source of Data (Producers)
CD
BLU-RAY
DVD
Receiver (Broker)
Destination of Data (Consumers)

Introduction to Apache Kafka

eazy bytes

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

Here are some key concepts and components of Kafka:



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

Introduction to Apache Kafka

eazy bytes

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

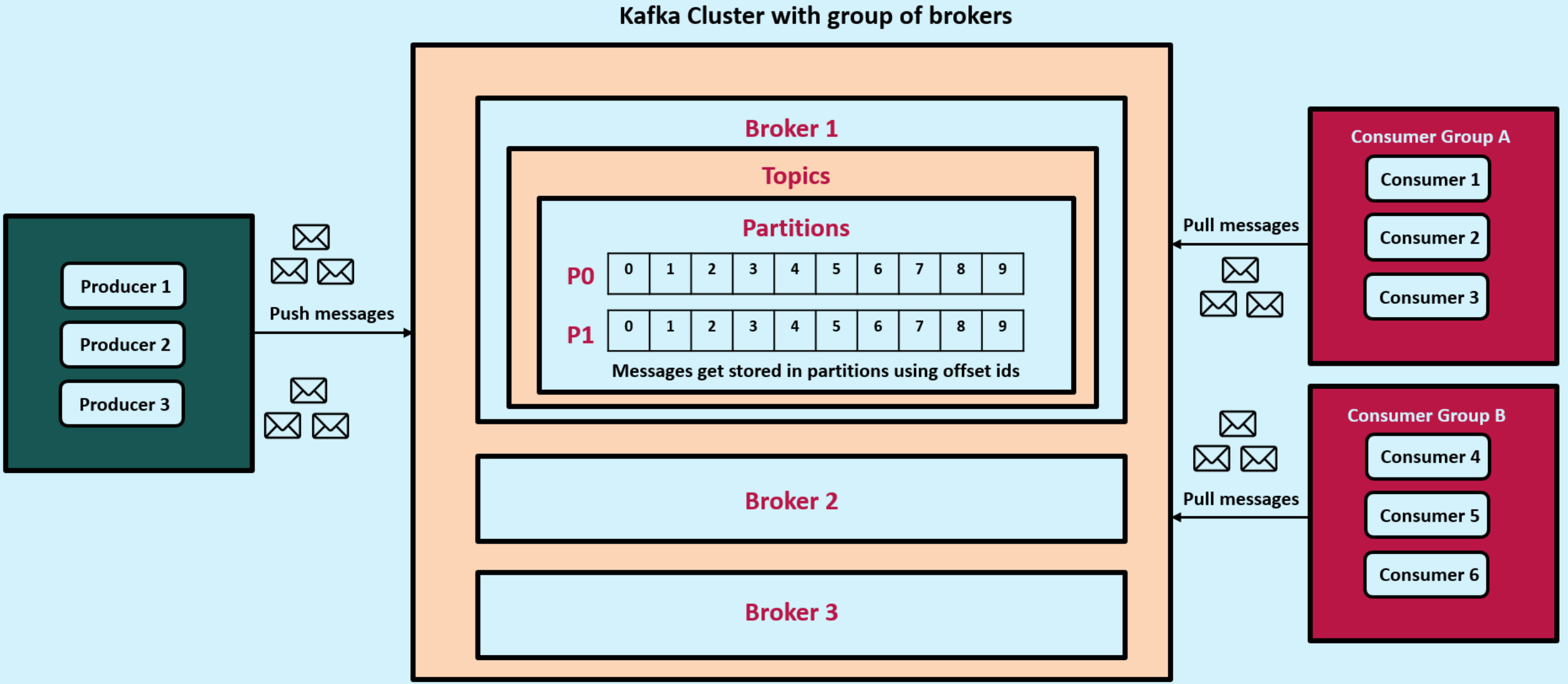
- **Consumers:** Consumers read messages from Kafka topics. They subscribe to one or more topics and consume messages by reading from specific partitions within those topics. Each consumer maintains its offset to track its progress in the topic.

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

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

Introduction to Apache Kafka

eazy bytes



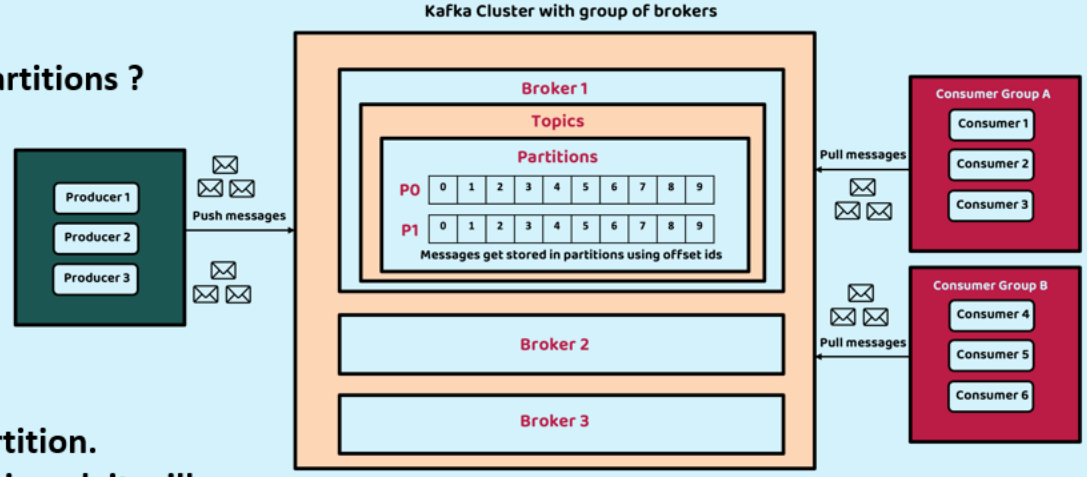
Introduction to Apache Kafka

eazy bytes

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

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

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



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

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

Producer side story

eazy bytes

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

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

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

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

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

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

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

Consumer side story

eazy bytes

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

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

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

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

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

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

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

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

Steps to use Apache Kafka in the place of RabbitMQ

eazy bytes

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

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

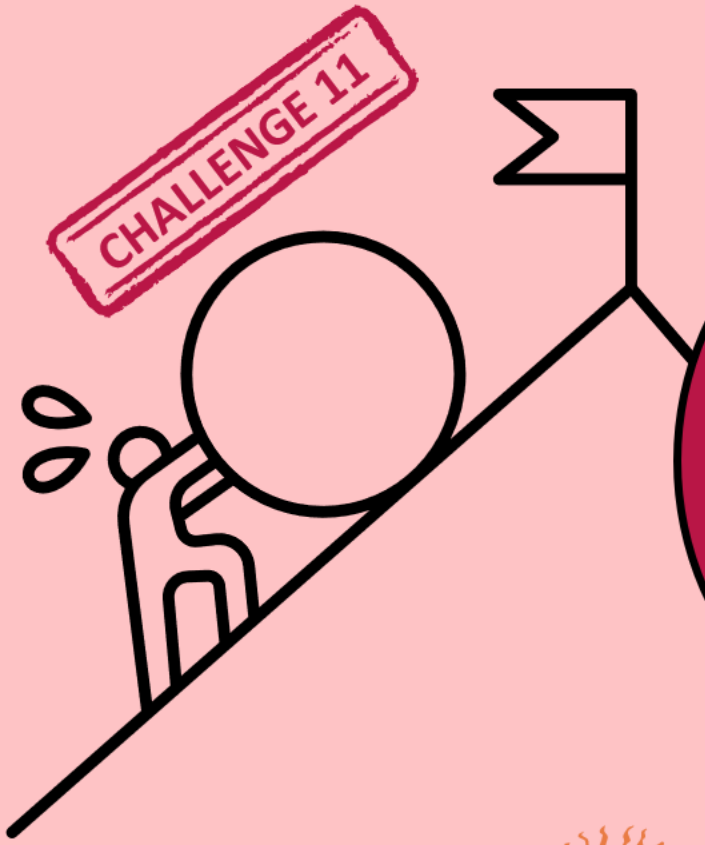
2. Add Kafka related properties inside the application.yml file of both accounts and message services

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

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

CONTAINER ORCHESTRATION

eazy bytes



AUTOMATING THE DEPLOYMENTS, ROLLOUTS & ROLLBACKS?

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

MAKING SURE OUR SERVICES ARE SELF-HEALING?

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

AUTOSCALING OUR SERVICES ?

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

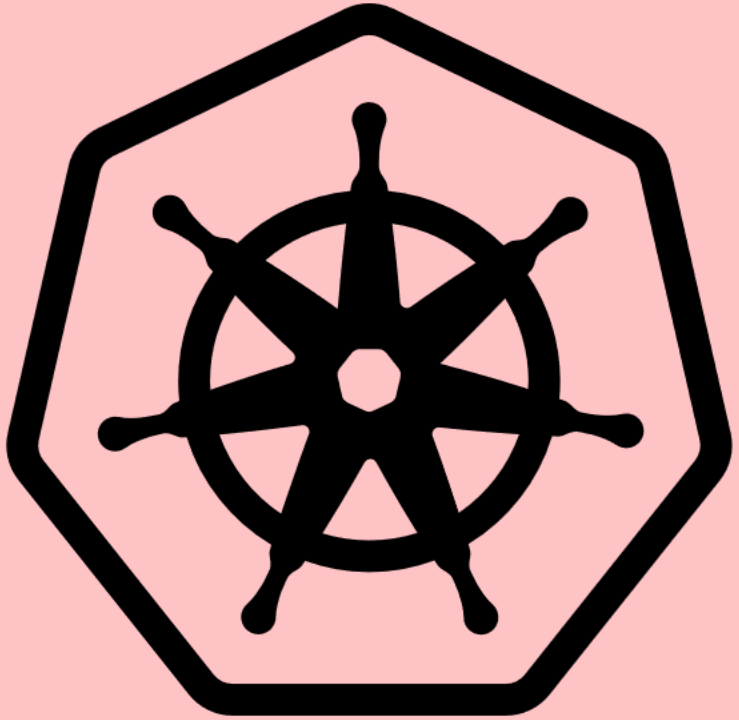
Kubernetes is an open-source container orchestration platform that automates the deployment, scaling, and management of containerized applications. It was originally developed by Google and is now maintained by the Cloud Native Computing Foundation (CNCF).

WHAT IS KUBERNETES ?

eazy bytes

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

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



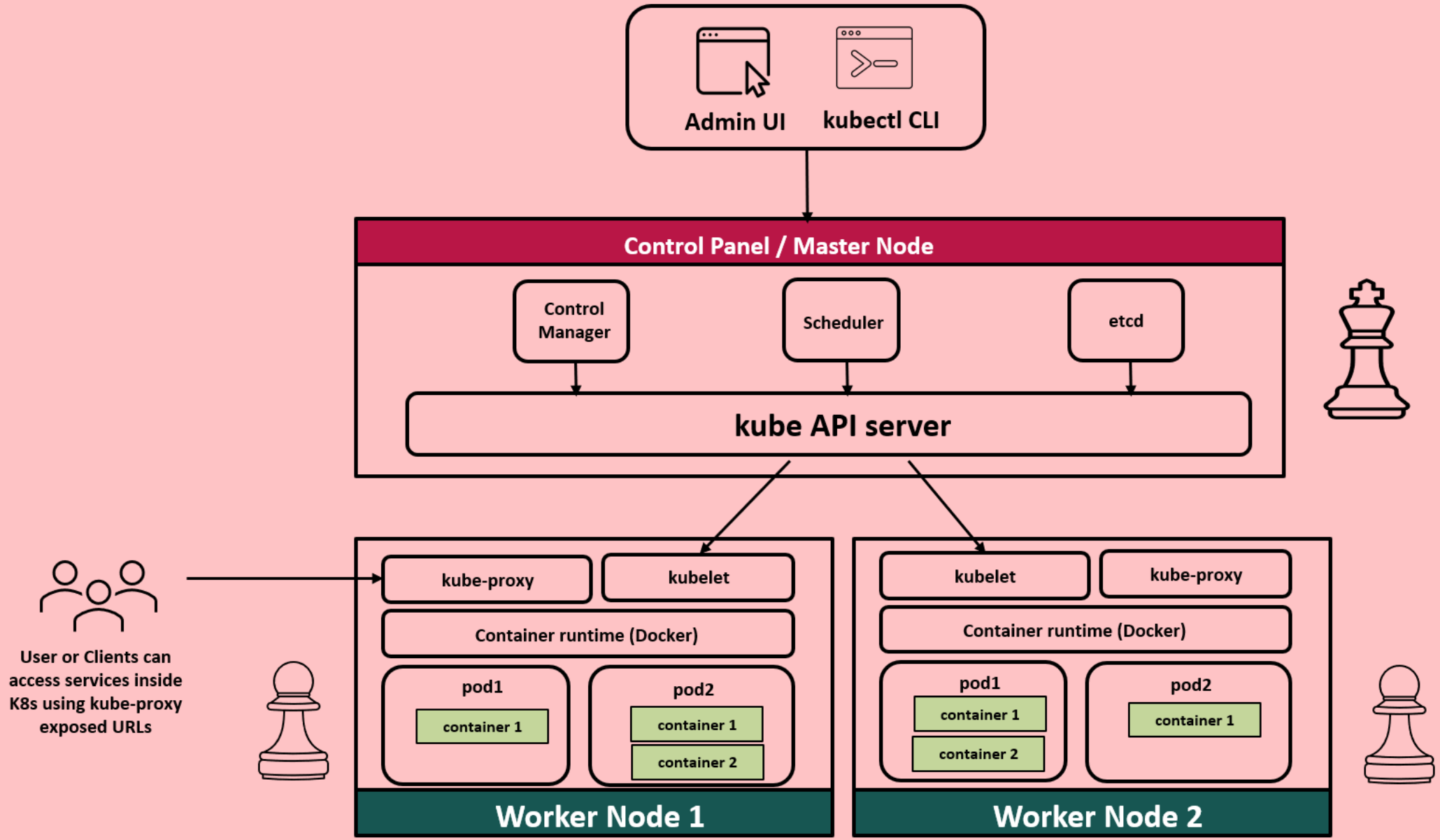
Kubernetes provides you with a framework to run distributed systems resiliently. It takes care of scaling and failover for your application, provides deployment patterns, and more. It provides you with:

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

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

KUBERNETES INTERNAL ARCHITECTURE

eazy bytes

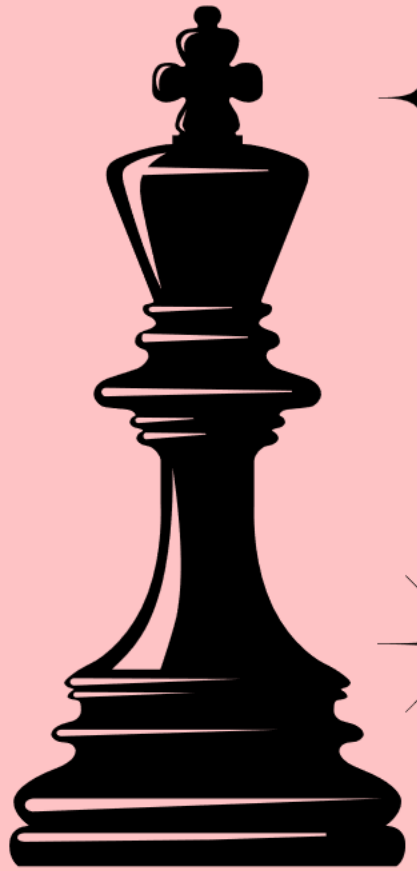


Components of Control Panel (Master Node)

eazy bytes

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

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



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

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

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

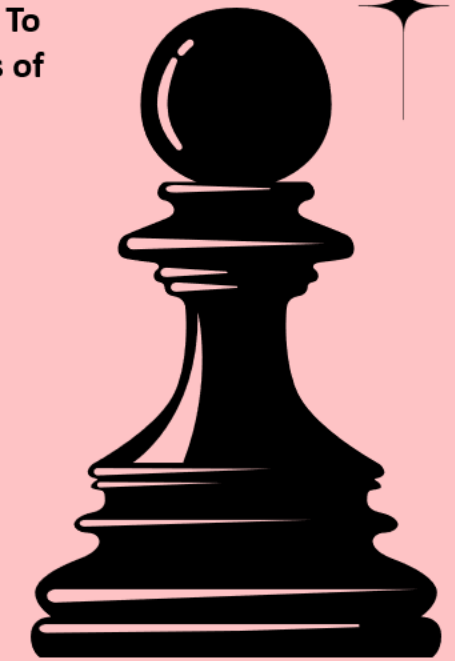
- **etcd** - etcd is a distributed key-value store that serves as the cluster's primary data store. It stores the configuration data and the desired state of the system, including information about Pods, Services, ReplicationControllers, and more. The API server interacts with etcd to read and write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

Components of Worker Node

eazy bytes

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

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



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

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

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

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

Kubernetes manifest file to create ConfigMap

eazy bytes

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

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

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

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

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

Kubernetes manifest file to deploy a Container

eazy bytes

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

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

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

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

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

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

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

Kubernetes manifest file to deploy a Container

eazy bytes

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

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

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

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

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

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

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

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

Kubernetes manifest file to create a service

eazy bytes

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

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

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

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

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

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

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

Kubernetes manifest file to create a service

eazy bytes

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

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

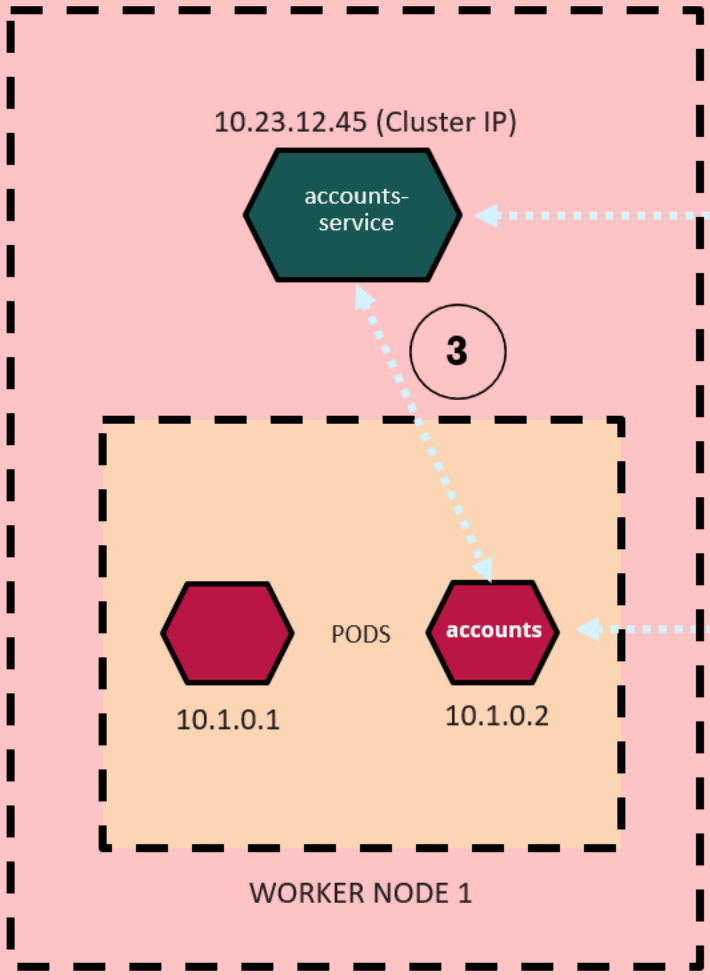
protocol: The protocol field specifies the protocol used for the service port. In this case, it's TCP.

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

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

How K8s deployment & services tied together

eazy bytes



2

1

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

3

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

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

Kubernetes Service Types

eazy bytes

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

ClusterIP Service

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

NodePort Service

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

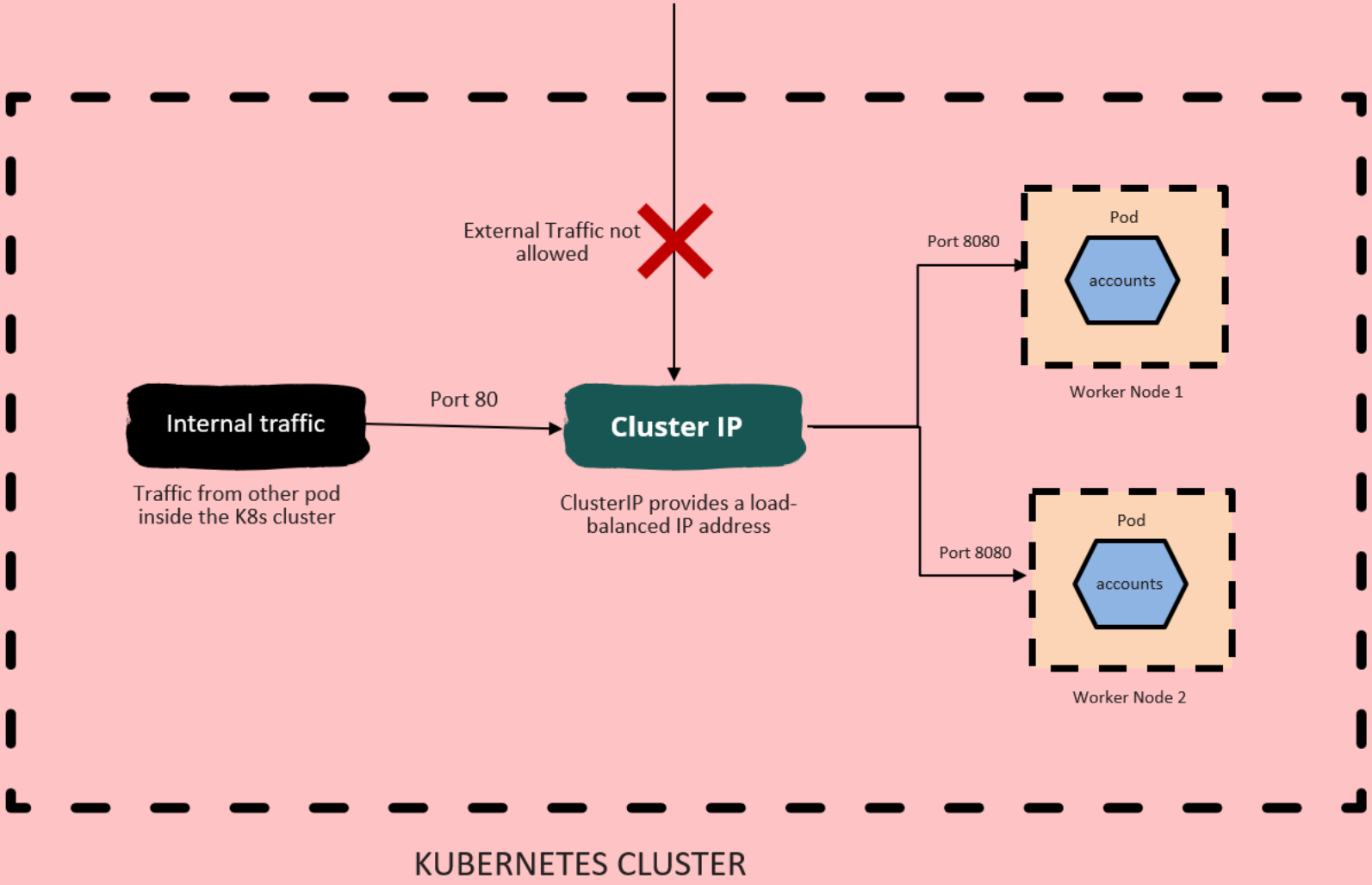
LoadBalancer Service

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

K8s CLUSTER IP SERVICE

eazy bytes

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



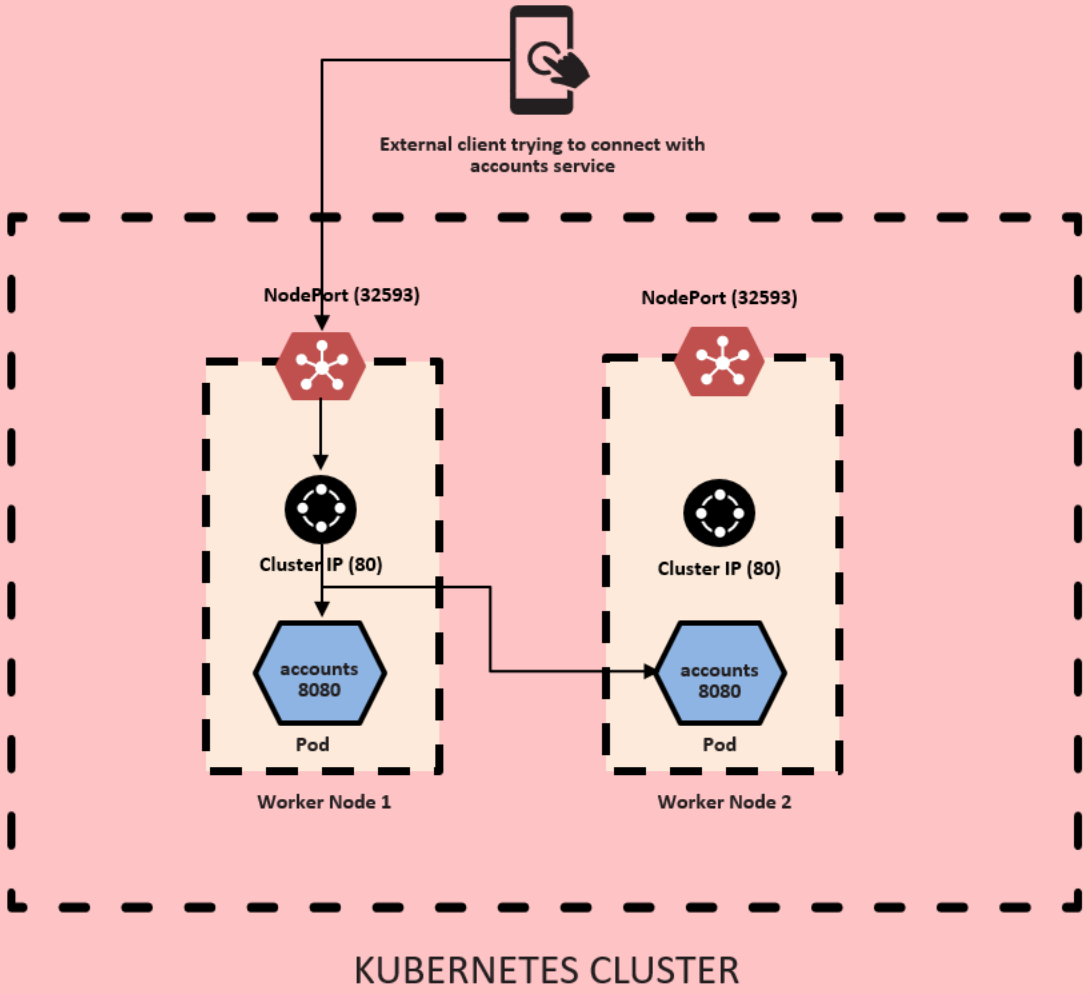

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: accounts-deployment
  labels:
    app: accounts
spec:
  replicas: 2
  selector:
    matchLabels:
      app: accounts
  template:
    metadata:
      labels:
        app: accounts
    spec:
      containers:
      - name: accounts
       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

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

K8s NODEPORT SERVICE

eazy bytes

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



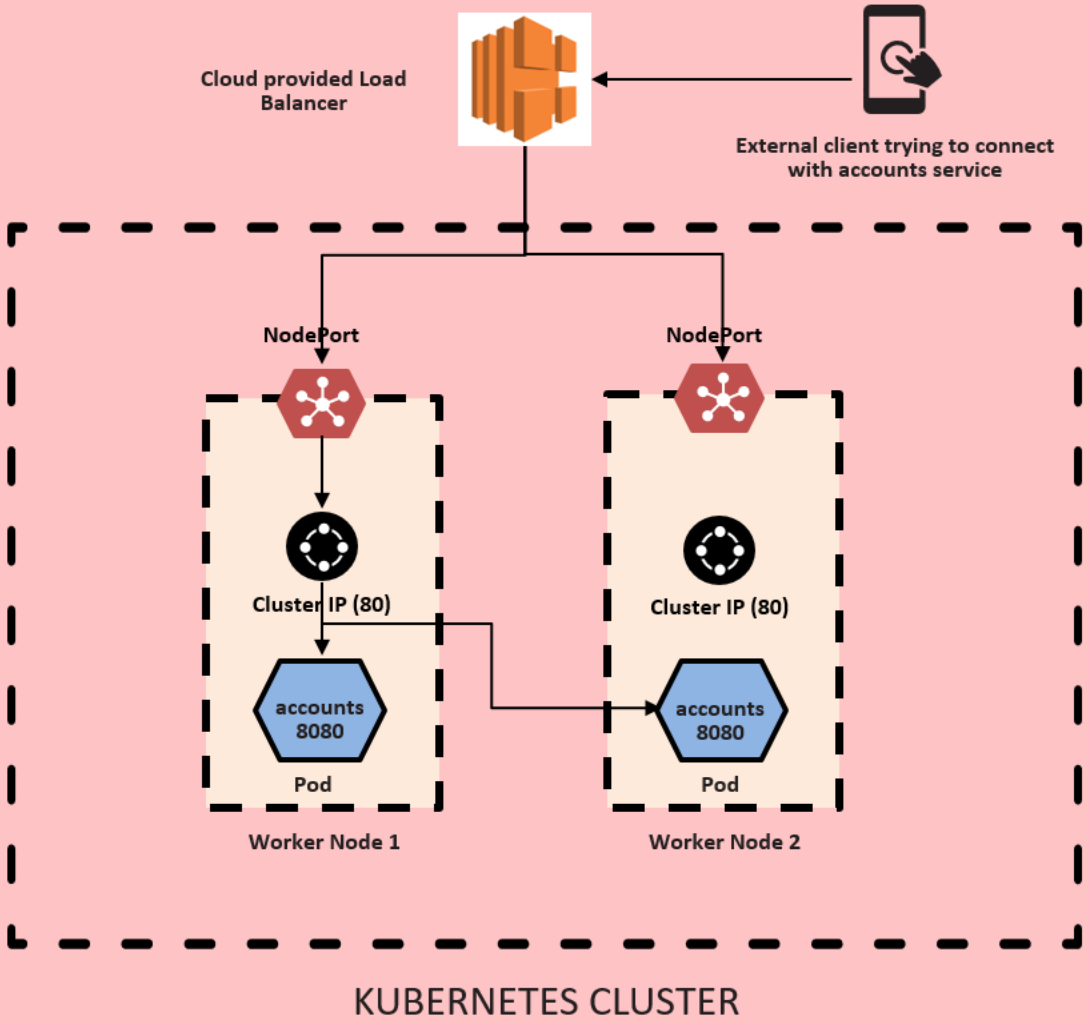

```yaml
apiVersion: v1
kind: Service
metadata:
  name: accounts-service
spec:
  selector:
    app: accounts
  type: NodePort
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
      nodePort: 32593
```

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

K8s LOADBALANCER SERVICE

eazy bytes

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




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

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

Introduction to Helm

eazy bytes

What is HELM ?

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

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

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

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

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

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

Problems that Helm solves

eazy bytes

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

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

accounts service manifest

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

loans service manifest

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

cards service manifest

Problems that Helm solves

eazy bytes

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

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

Helm service template

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

values.yaml

Problems that Helm solves

eazy bytes

HELM SUPPORTS PACKAGING OF YAML FILES

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

HELM SUPPORTS EASIER INSTALLATION

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

HELM SUPPORTS RELEASE/VERSION MANAGEMENT

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

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

Helm chart structure

eazy bytes

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

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

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

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

templates folder contains the manifest template yaml files

Important Helm commands

eazy bytes

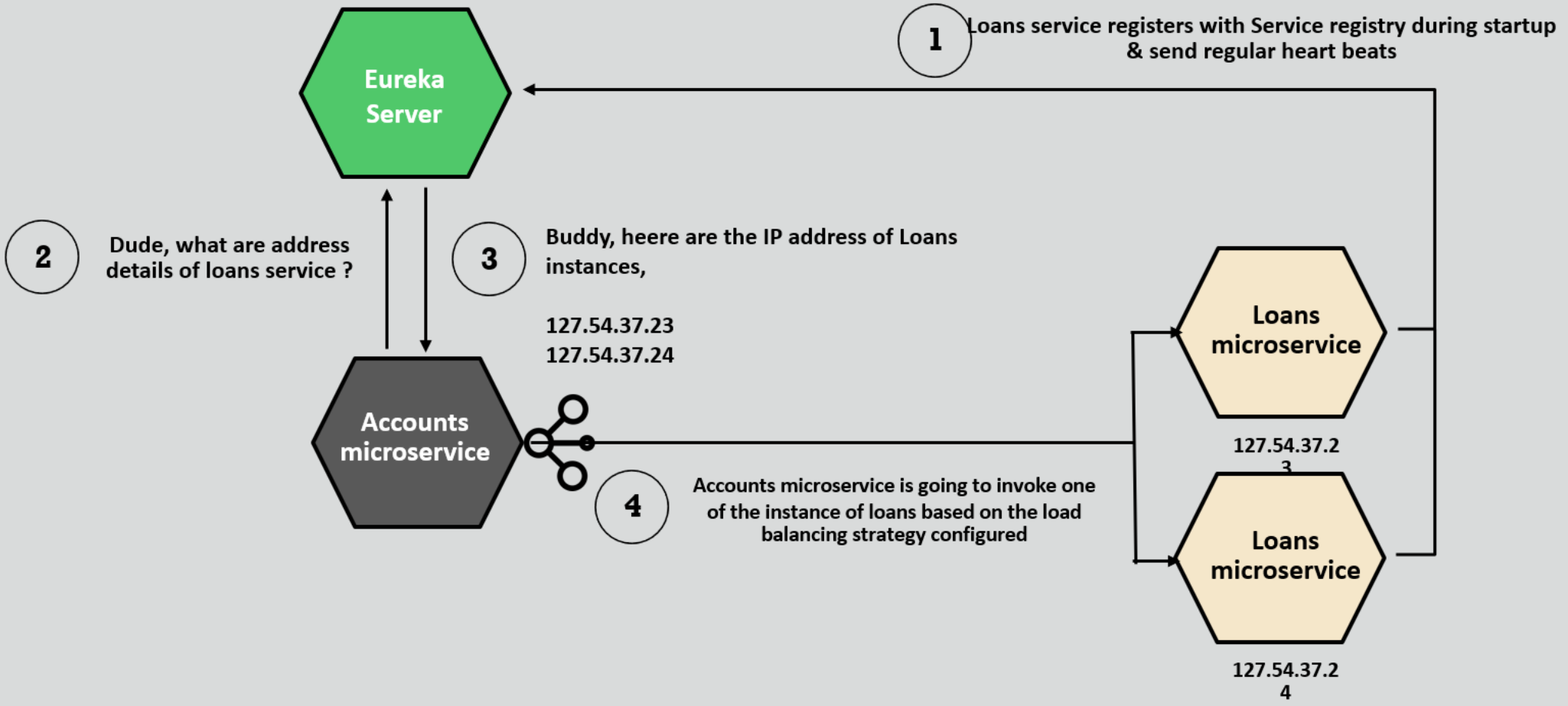
Helm commands

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

Client-side service discovery and load balancing

eazy bytes

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

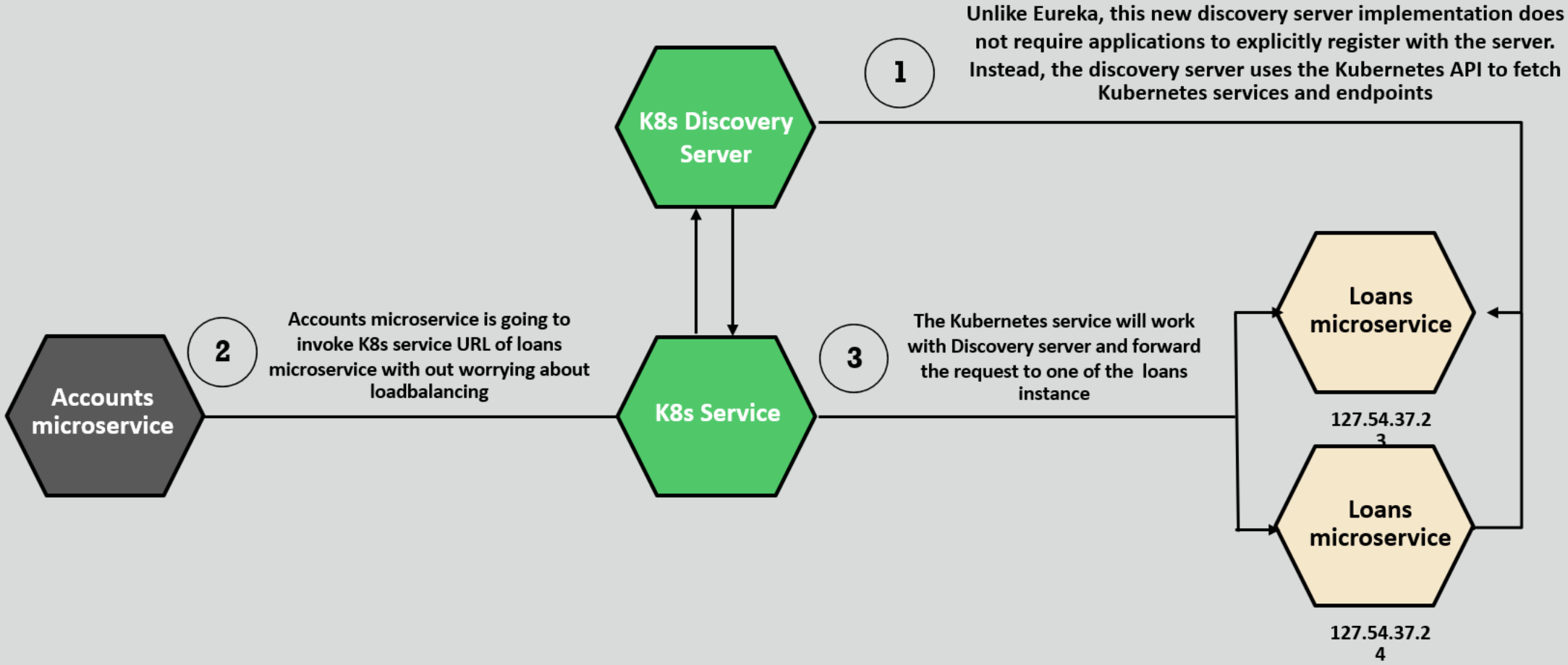


Server-side service discovery and load balancing

eazy bytes

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

Below figure illustrates the workflow of this process,



Kubernetes support by Cloud providers

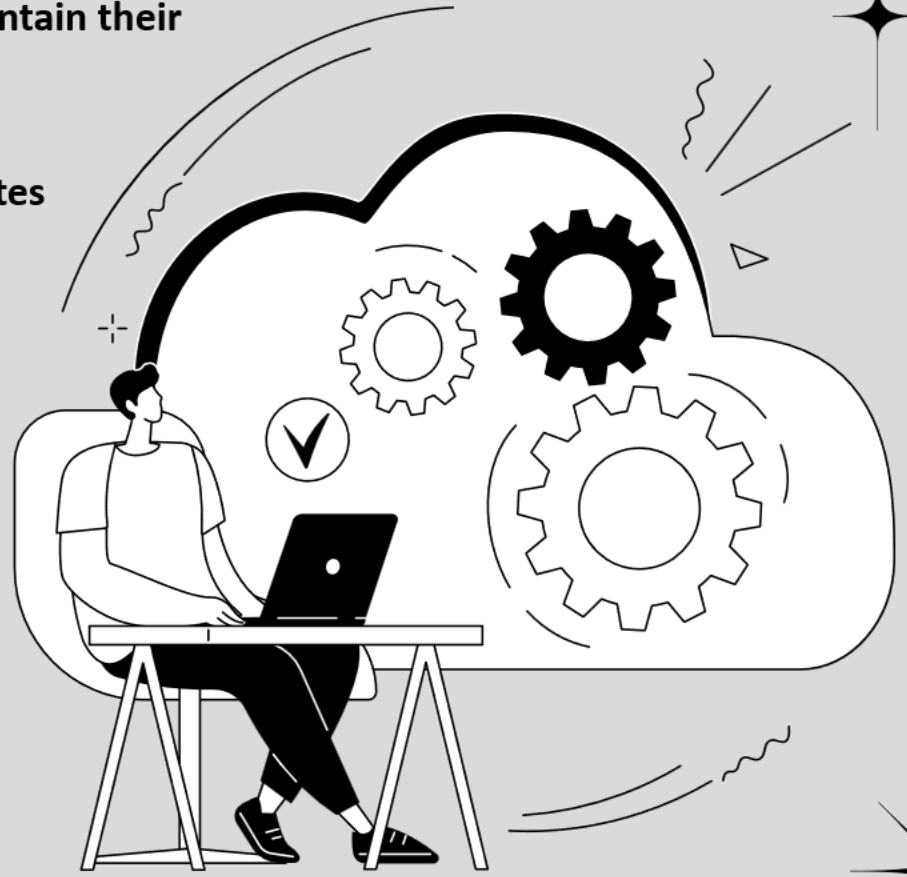
eazy bytes

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

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

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

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



Kubernetes Ingress

eazy bytes

What is Ingress ?

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

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

What is Ingress Controller ?

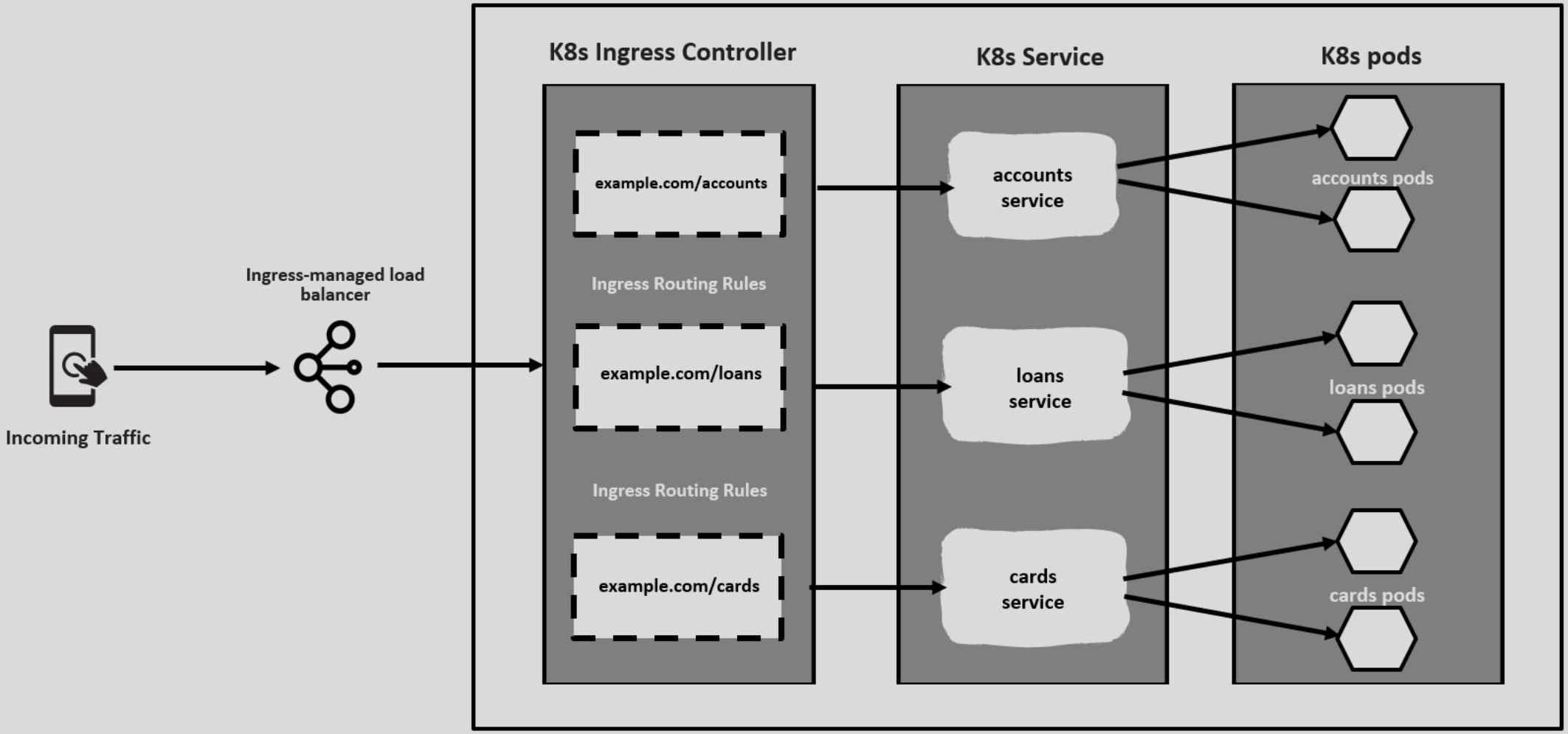
On its own, the Ingress resource doesn't do anything. You need to have an Ingress controller installed and configured in your cluster to make Ingress resources functional. Popular Ingress controllers include Nginx Ingress, Traefik, and HAProxy Ingress. The controller watches for Ingress resources and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

Full list of Ingress Controllers available in the below URL,

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

Kubernetes Ingress

eazy bytes



Traffic flow in K8s Cluster with Ingress controller

Kubernetes Ingress

eazy bytes

Why Use Ingress ?

Ingress provides several benefits:

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

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

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

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

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

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

Kubernetes Ingress

eazy bytes

Ingress Controllers vs. Service Type LoadBalancer

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

Types of traffic handled by Ingress Controller

Ingress traffic: Traffic entering a Kubernetes cluster

Egress traffic: Traffic exiting a Kubernetes cluster

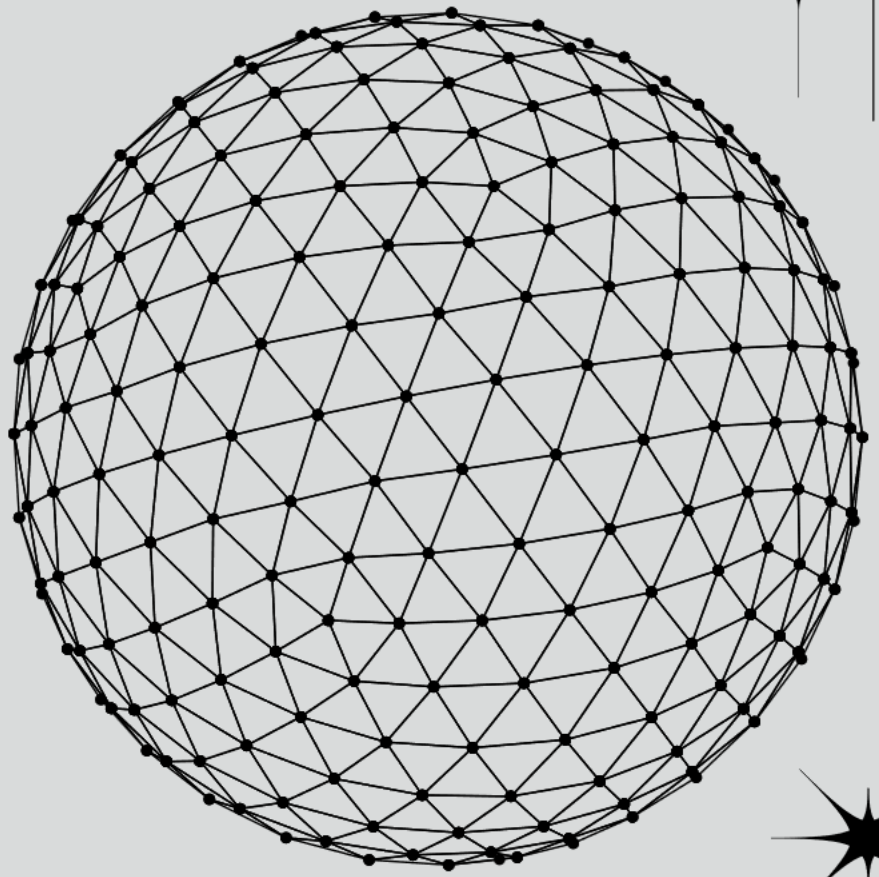
North-south traffic: Traffic entering and exiting a Kubernetes cluster (also called ingress-egress traffic)

Who handles service-service traffic ?

eazy bytes

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

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



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

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

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

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

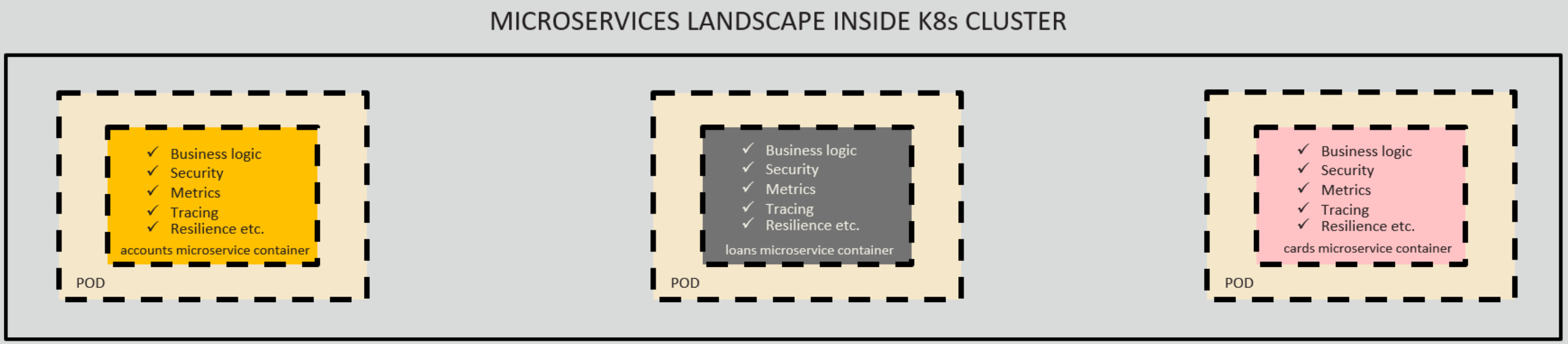
Fault tolerance: This ensures that services can continue to function even if some of their dependencies fail.

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

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

Problem that Service Mesh trying to solve

eazy bytes



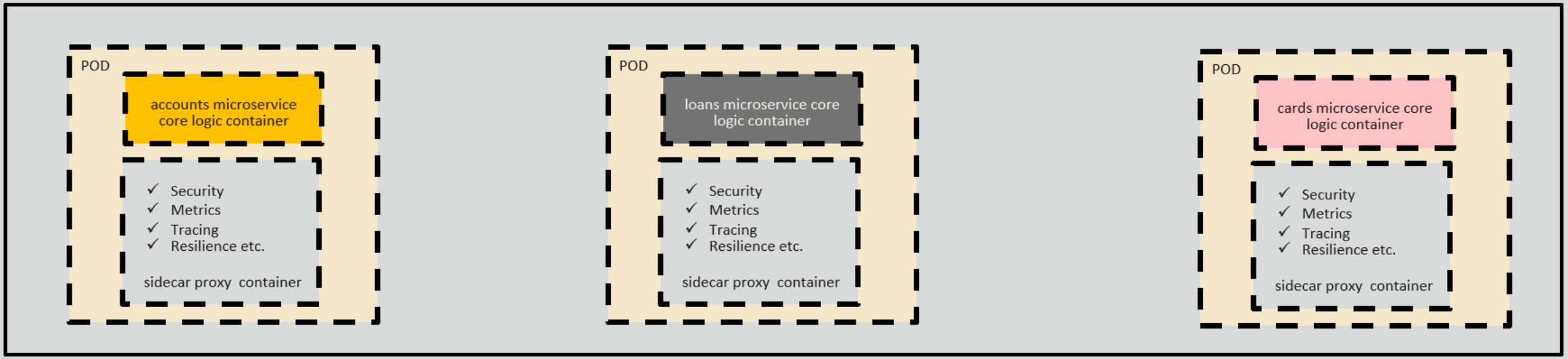
Each microservice has lot of code/configurations related to Security, tracing etc. which is not related to business logic. All the extra code/configurations that is present inside each microservice will make them complex.

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

How Service Mesh makes our life easy while building microservices

eazy bytes

MICROSERVICES LANDSCAPE INSIDE K8s CLUSTER



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

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

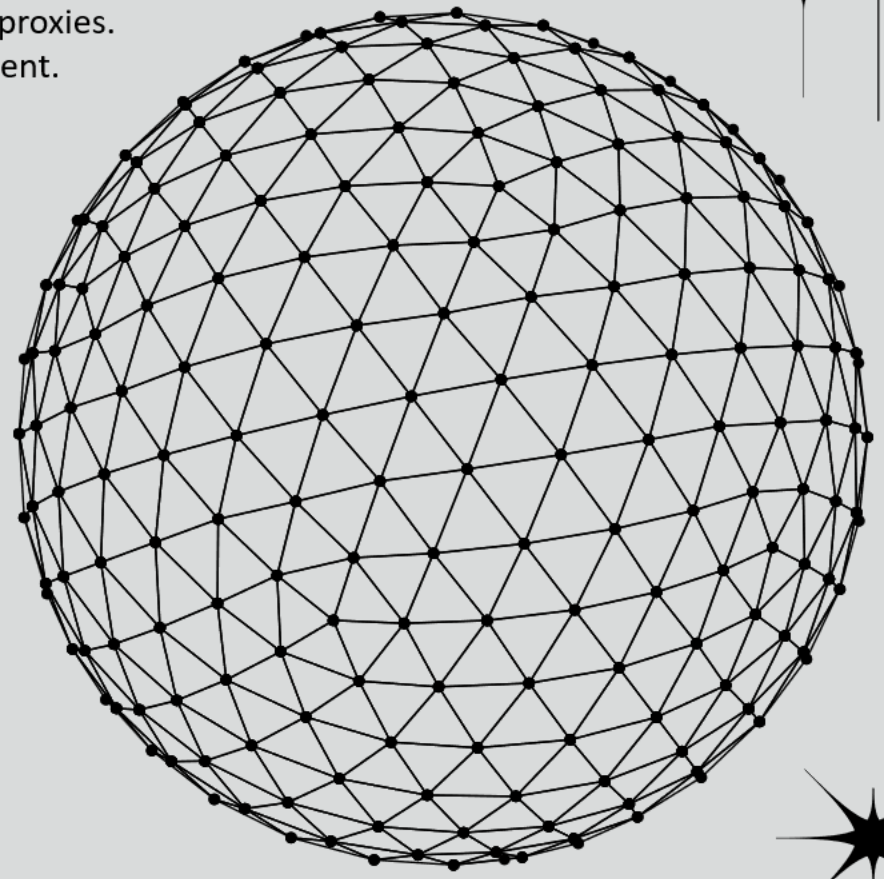
Service mesh components

eazy bytes

A service mesh typically consists of two components:

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

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



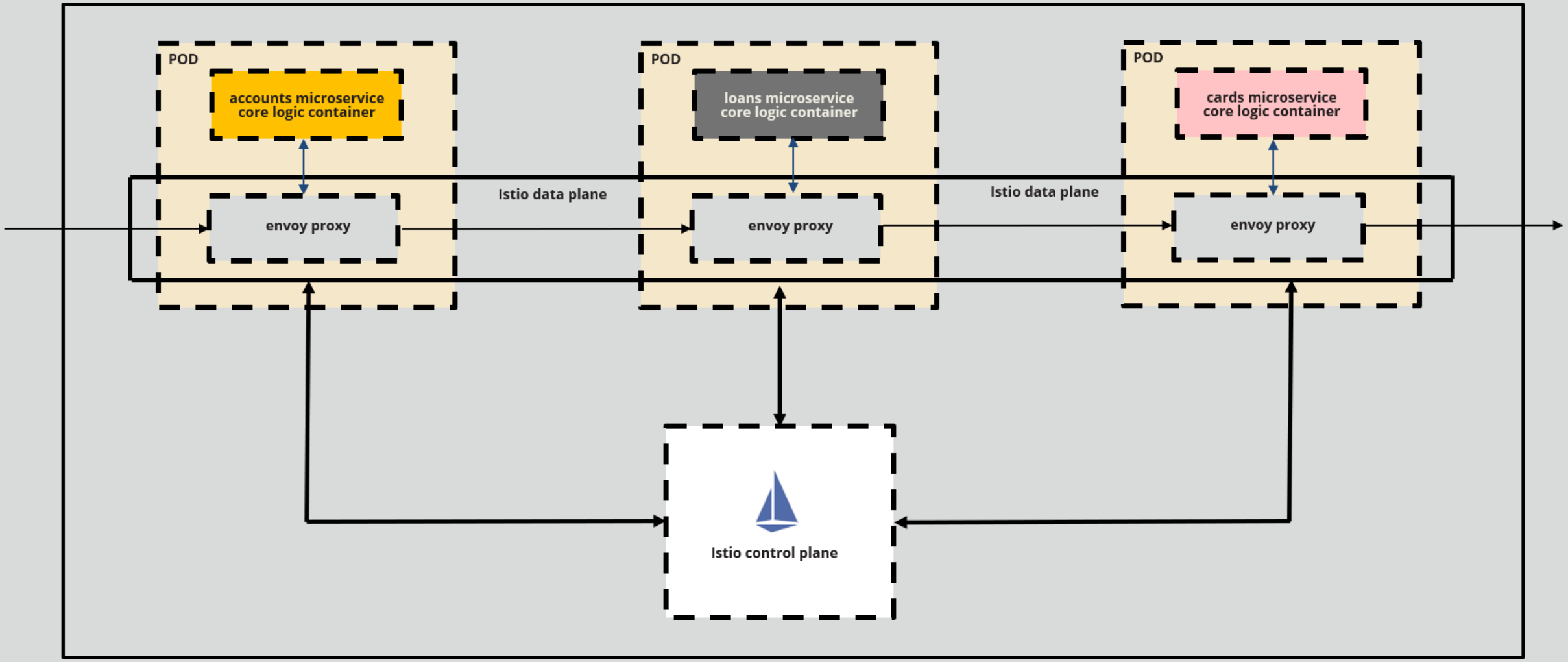
Here are some of the popular service meshes:

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

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

Istio service mesh components

eazy bytes



Introduction to mutual TLS (mTLS)

eazy bytes

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

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

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

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

What is TLS?

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

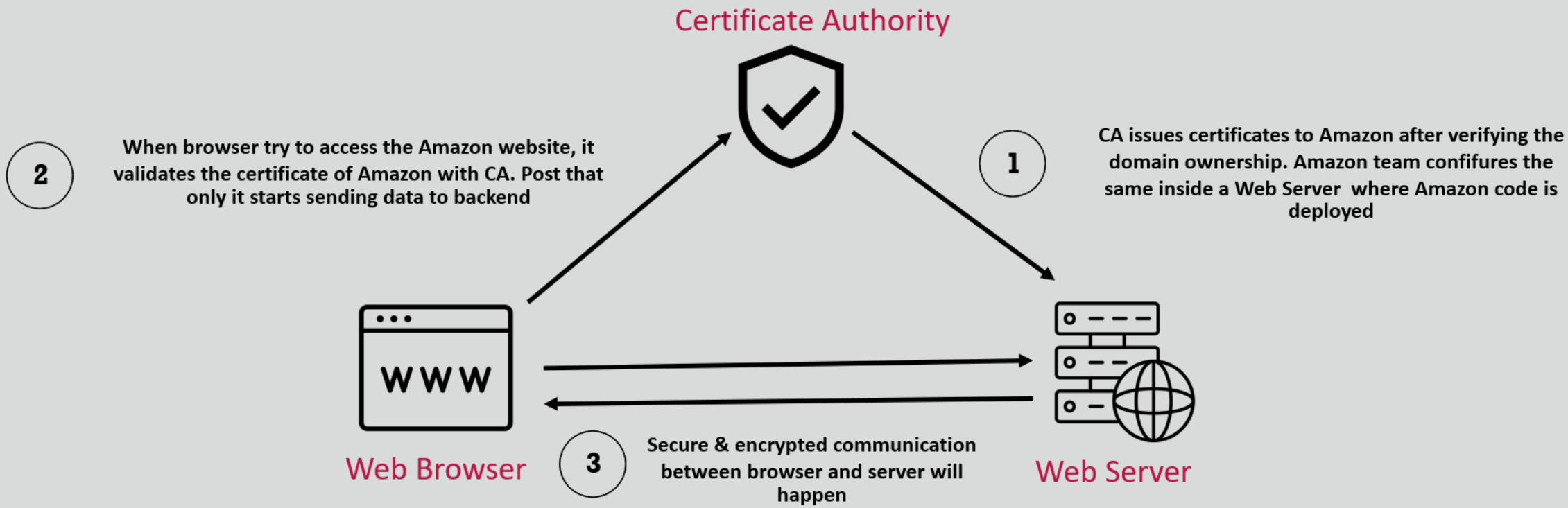
TLS is the direct successor to SSL, and all versions of SSL are now deprecated. However, it's common to find the term SSL describing a TLS connection. In most cases, the terms SSL and SSL/TLS both refer to the TLS protocol and TLS certificates.

How does TLS works ?

eazy bytes

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

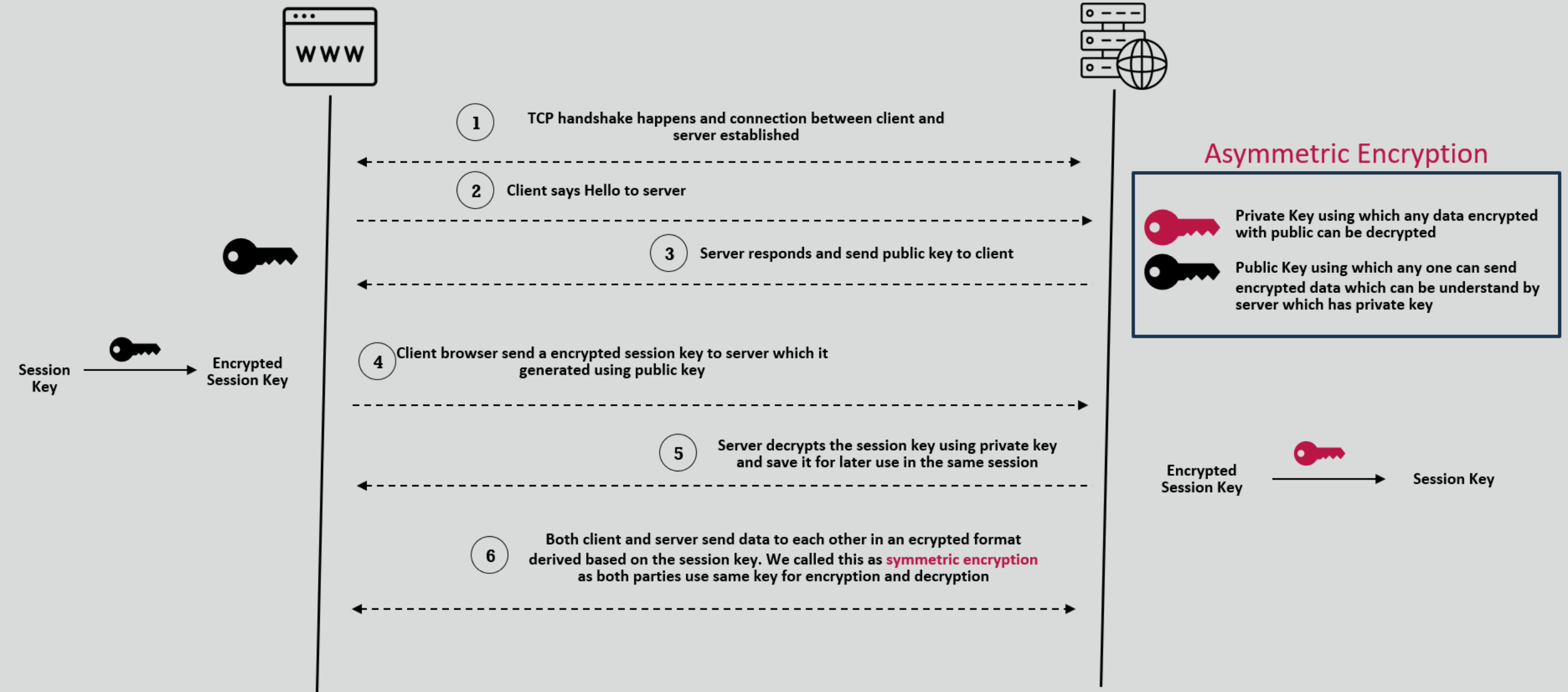
Even if we've never visited www.amazon.com before, our web browsers inherently trust the site's identity right from our initial visit. This trust is enabled by the involvement of trusted third parties (TTPs). In the context of TLS, these TTPs are known as certificate authorities (CAs), which generate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



How does TLS works ?

eazy bytes

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



How Is mTLS Different from TLS ?

eazy bytes

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

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

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

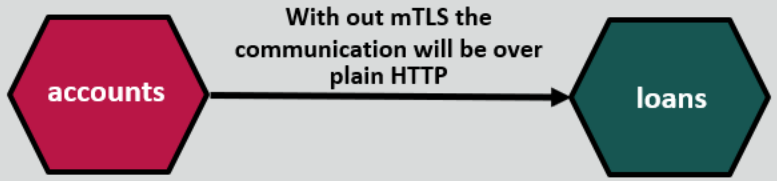
Yet, managing many certificates in mTLS can become challenging, which is where automatic mTLS through a service mesh helps ease certificate management complexity.

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

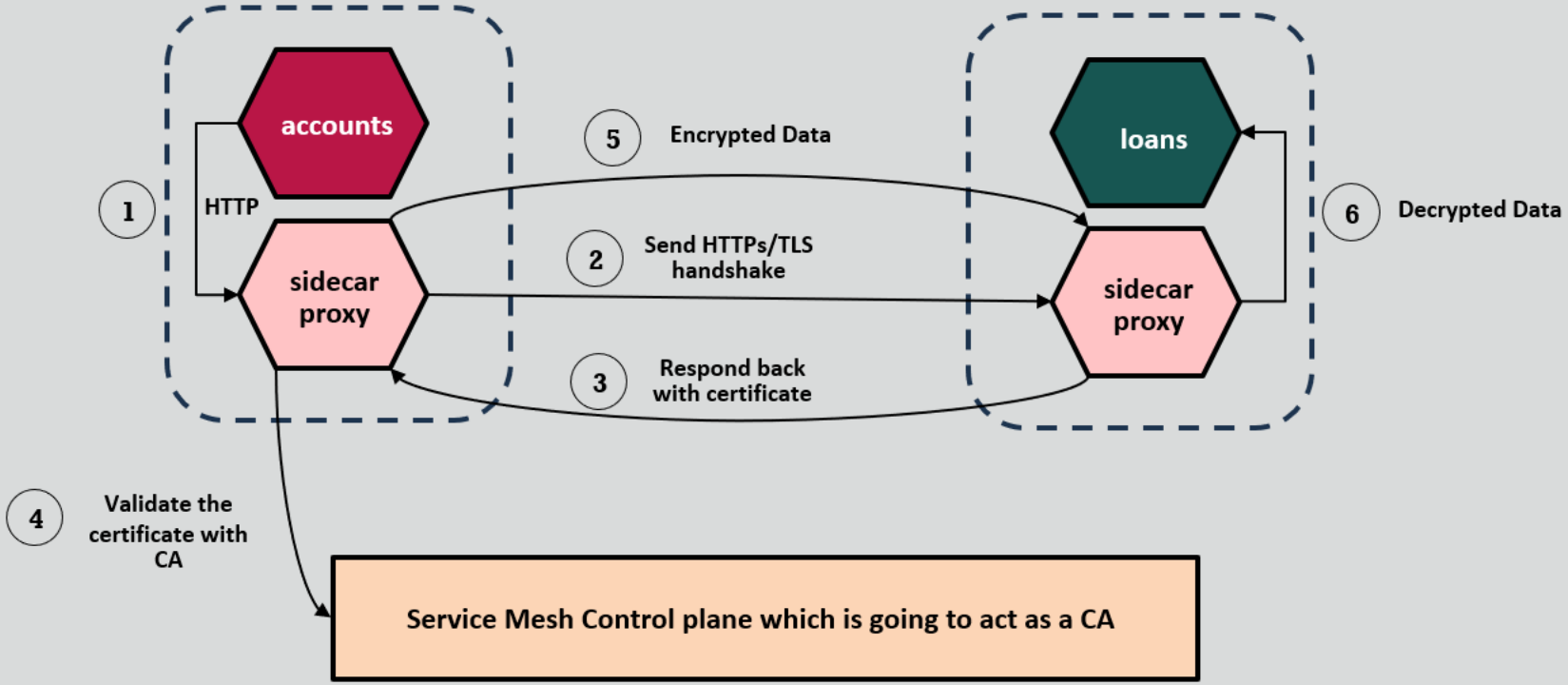
How does mTLS works ?

eazy bytes

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



Sample flow of mTLS between two microservices



Why use mTLS?

eazy bytes

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

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

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

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

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

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

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

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

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

Optimizing Microservices Development with Spring Boot BOM

eazy bytes

What is a BOM in Maven?

BOM is a dependency management mechanism that provides a central place to define the versions of dependencies (and their transitive dependencies). It prevents version conflicts when different parts of your system use the same libraries but specify different versions.

With a BOM, you can manage versions centrally in one place, and downstream projects just need to import the BOM to automatically align their versions.

A BOM (Bill of Materials) in the context of Maven and Spring Boot is a special kind of POM (Project Object Model) that manages the versions of a set of related dependencies. It is particularly useful for ensuring consistency and avoiding version conflicts across multiple modules in a microservices architecture.

Why Use BOM in Microservices?

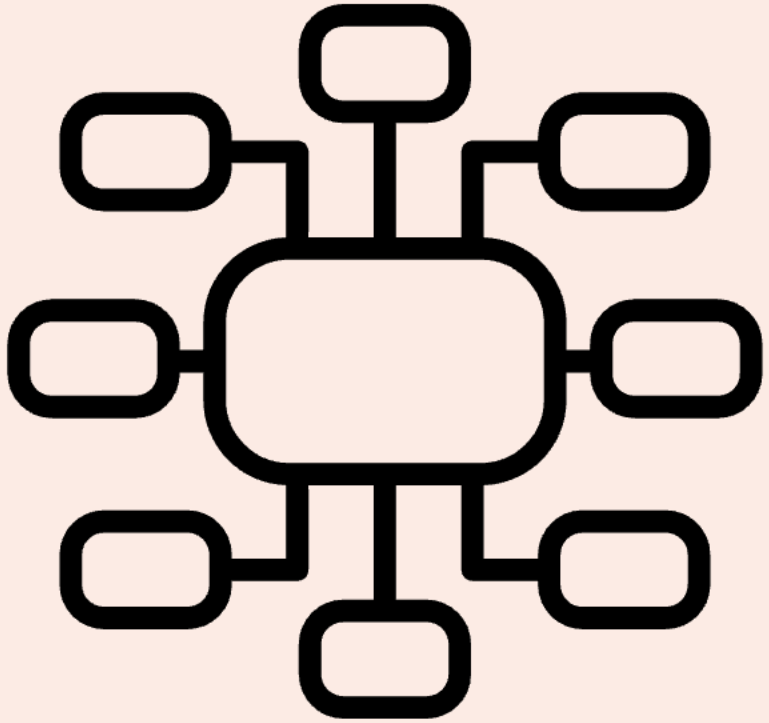
In microservices architecture, managing shared libraries across multiple services can quickly become complex. Each service might independently manage dependencies, leading to version conflicts and increased maintenance effort. The BOM (Bill of Materials) addresses this challenge by:

- Ensuring version consistency across all microservices
- Streamlining dependency management
- Simplifying the process of upgrading libraries across all services

Shared Libraries in Microservices

eazy bytes

In large microservice projects, code duplication—like utility classes and configurations—becomes inevitable. Bugs found in one service need to be fixed across multiple places, and different developers may introduce inconsistent fixes, leading to further fragmentation. Let's explore some approaches for code sharing in microservices to address this issue.



One approach is to create a shared Maven project containing all common dependencies for the microservices. However, this can lead to larger JAR files due to unused dependencies and unnecessary bean creation.

Another approach is to split the shared code into multiple smaller libraries. While this avoids the issues of unused dependencies, it introduces new challenges. Updating several libraries simultaneously requires multiple pull requests and publishing multiple versions, complicating the management process.

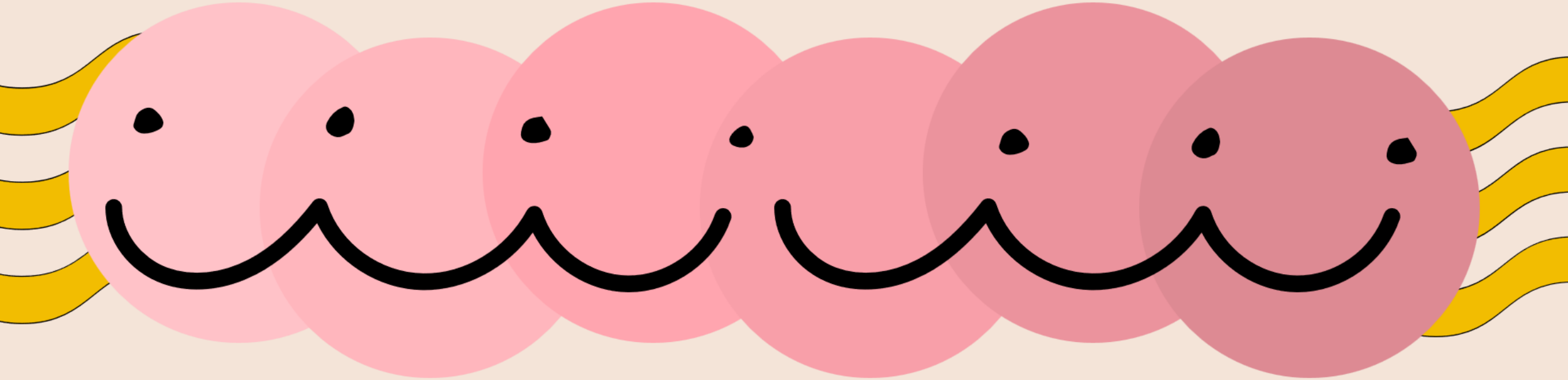
A multi-module project in Maven is a project that consists of multiple submodules, each with its own POM file, but sharing a common parent POM. In microservices, a multi-module project can be used to share common code between services without duplicating it.

In a microservices architecture, sharing code or libraries can be a debated topic. While there are scenarios where shared code might make sense, it's not always recommended or considered best practice. So do your due dilligence

eazy bytes

CONGRATULATIONS

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



A BIG THANK YOU

HOPING FOR OUR PATHS TO CROSS AGAIN