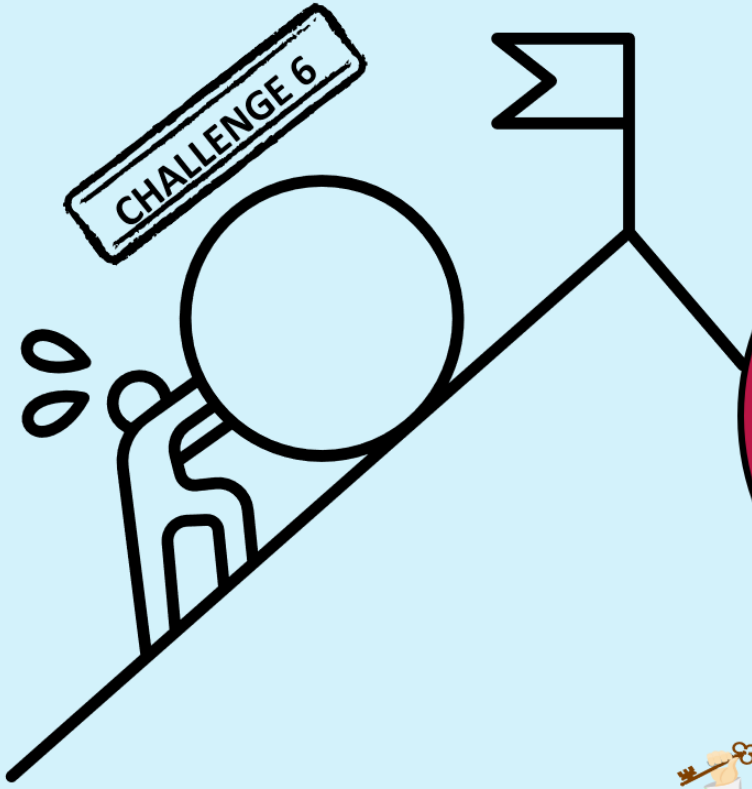
ROUTING, CROSS CUTTING CONCERNS IN MICROSERVICES

eazy bytes



HOW DO WE MAINTAIN A SINGLE ENTRYPOINT INTO MICROSERVICES NETWORK ?

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

HOW DO WE HANDLE CROSS CUTTING CONCERNS?

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

HOW DO WE ROUTE BASED ON CUSTOM REQUIREMENTS

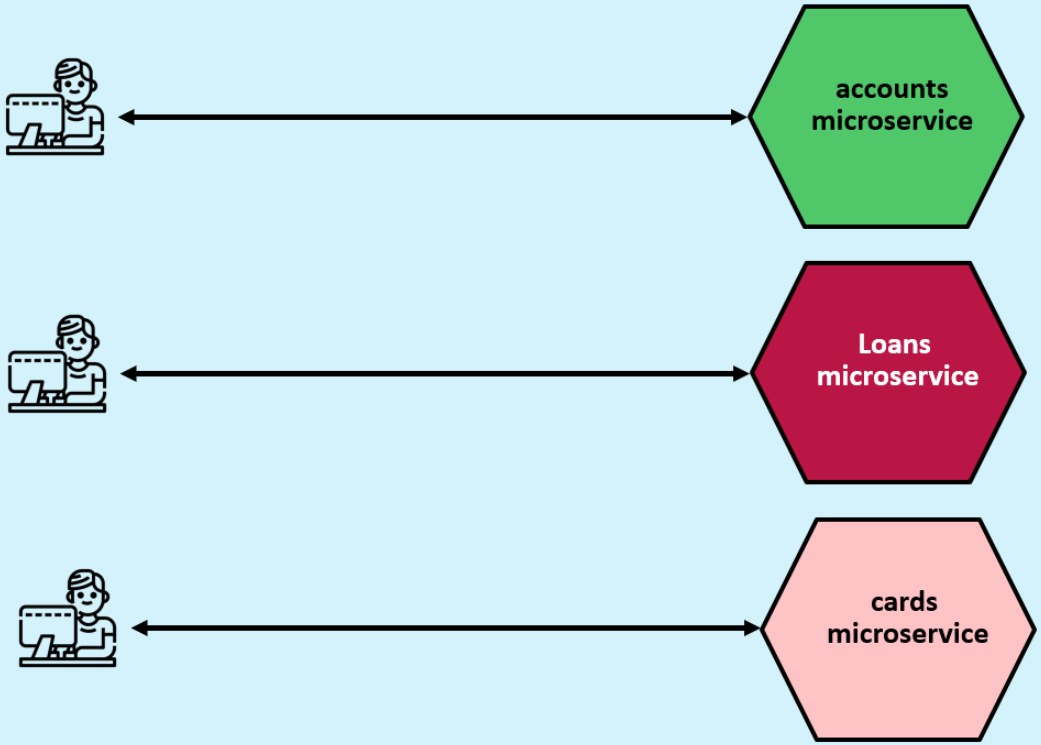
How to provide dynamic routing capabilities which allows to define routing rules based on various criteria such as HTTP headers, request parameters etc. inside microservices network ?

These challenges in microservices can be solved using a

Edge server

ROUTING, CROSS CUTTING CONCERNS IN MICROSERVICES

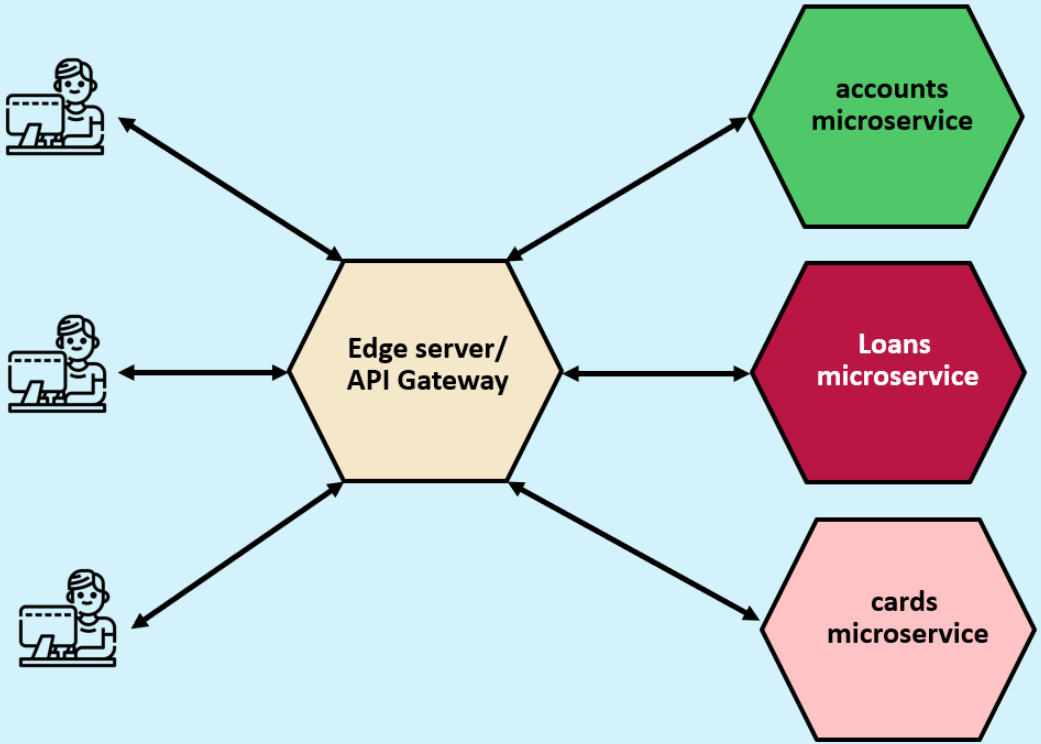
eazy bytes



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

ROUTING, CROSS CUTTING CONCERNS IN MICROSERVICES

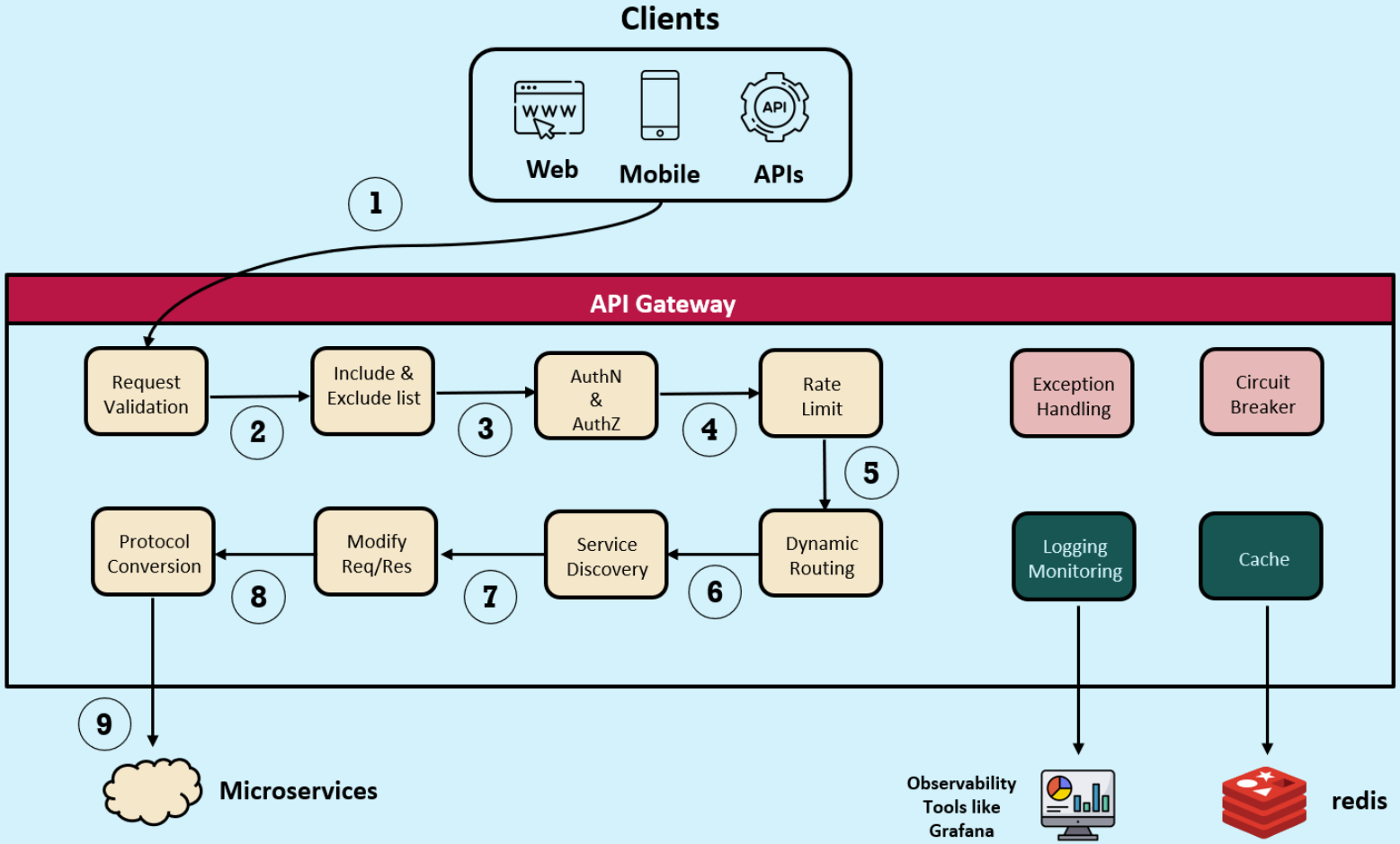
eazy bytes



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

Few important tasks that API Gateway does

eazy bytes



Spring Cloud Gateway

eazy bytes

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

Here are the key aspects of Spring Cloud Gateway,

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

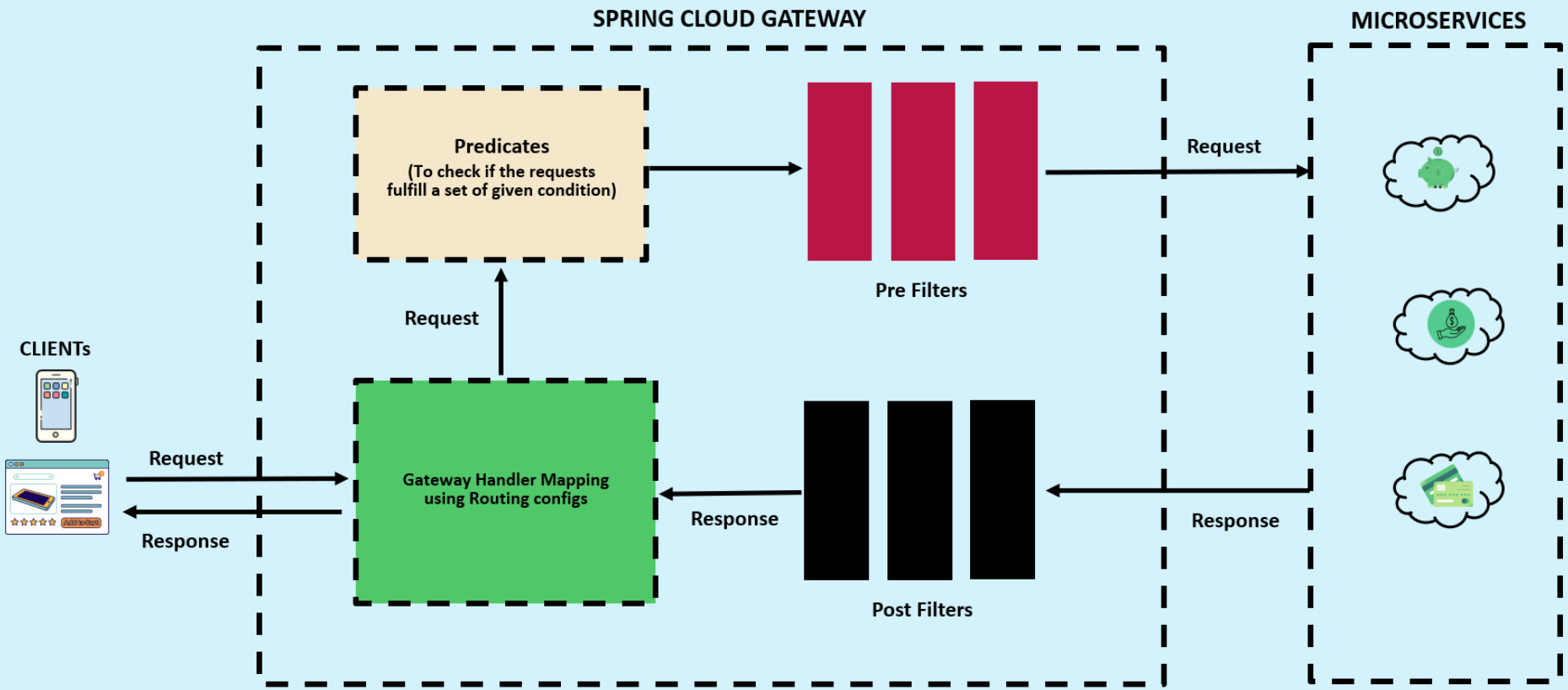
Spring Cloud Gateway is the preferred API gateway compared to zuul. Because Spring Cloud Gateway built on Spring Reactor & Spring WebFlux, provides circuit breaker integration, service discovery with Eureka, non-blocking in nature, has a superior performance compared to that of Zuul.

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

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

Spring Cloud Gateway Internal Architecture

eazy bytes



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

Steps to create Spring Cloud Gateway

eazy bytes

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

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

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

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

Steps to create Spring Cloud Gateway

eazy bytes

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

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

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

- As we can see the code from gatewayserver application, we have defined the routes and some default filters for routing the request to specific API, also we can see a custom filter has been added which is ".addResponseHeader", similarly we can added multiple header, redirect the request to different URL, have caching mechanism and much more.

- But there might be a scenario where we don't find a filter that meets the business needs and in that case we have to go with custom filter.

- Creating custom filters

let's take a business scenario where as soon as our gateway server receives an external traffic or external request, it has to generate a correlation ID. So here ID is a randomly generated value and the same correlation ID we want to send for all the further micro-services that our request is going to travel. Maybe the request will travel from Gateway server to accounts, accounts to loans and cards, then we want to make sure the same trace ID or the correlation ID to be sent to all the respective micro-services. And using the same correlation ID or the trace ID,

we can add some logger statements inside my micro-services. While we are sending the response back to the client, we can add the same correlation ID inside the response header so that in future, whenever my client has some issues with a particular request saying that the data is not correct or there is an exception. So using that correlation ID of we can look at the logs and identify up to which micro-service the request has traveled and if there is an exception then we can identify under which micro-service the exception received.