MICROSERVICES- SPRING BOOT

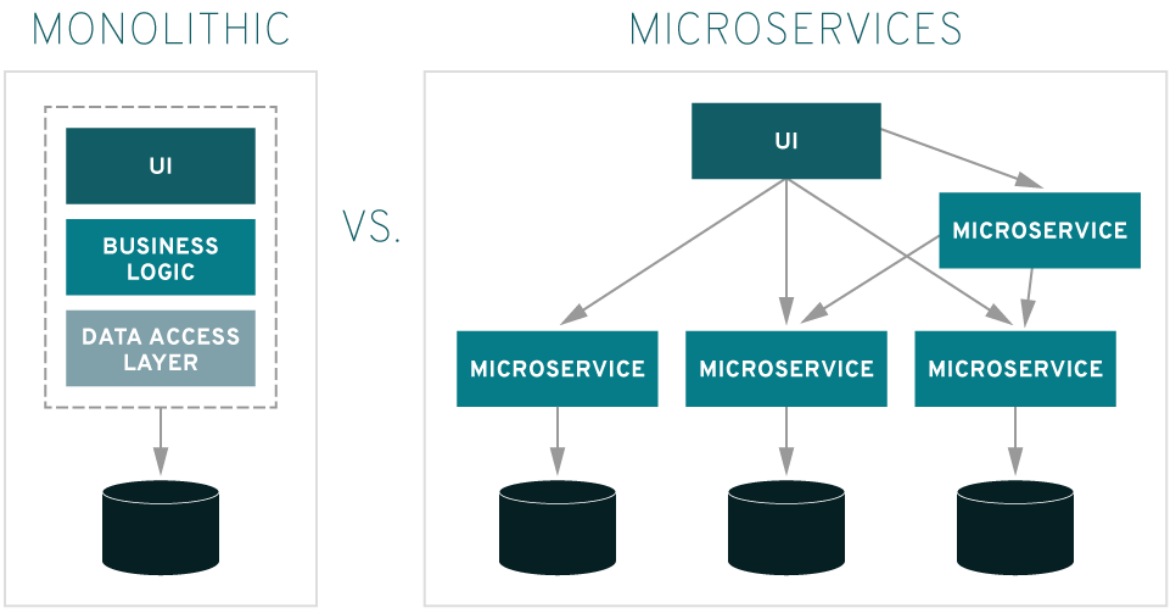
#1.Introduction

Microservices are an architectural style where applications are developed as a collection of small, loosely coupled, independently deployable services.

Characteristics of Microservices

Small, focused on doing one thing well
- Independently deployable
- Organized around business capabilities
- Decentralized data management
- Communication via lightweight protocols (typically HTTP/REST)

Comparison with Monolith
- Monolith: tightly coupled, single codebase
- Microservices: loosely coupled, multiple independent services



Increased complexity in managing multiple services

- Distributed System Issues like Latency, load balancing, network reliability, and consistency
- Managing transactions and consistency across services

• Handling communication protocols (REST, gRPC, messaging)
• Monitoring and Logging needs centralized monitoring and logging solution.

Why use Microservices

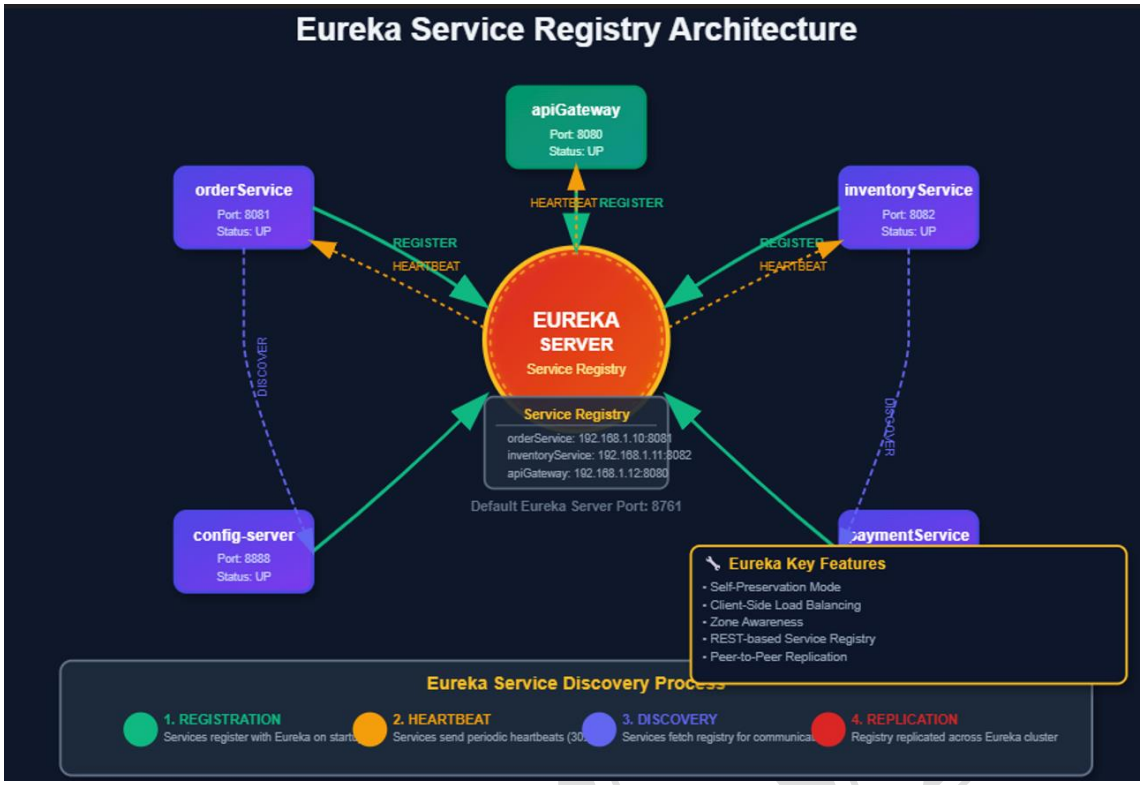
Scale services independently based on demand (e.g., scale only payment service during high traffic)
• Services can be written in different programming languages, suited to specific tasks
• Small teams can work on different services simultaneously, reducing time to market
• Failure in one service doesn't bring down the entire system
• Each service can be updated, deployed, and scaled independently
• Teams can work on different services without affecting each other

#2.What is Service Discovery

In a microservices architecture, each microservice is a standalone application with specific business functionality. Since these microservices need to communicate with each other to function as a complete application, they need to know each other's network locations. **Service Discovery comes into play here, maintaining a record of these services' locations, helping them find each other, and enabling communication.**

Spring Cloud Eureka

Eureka is a REST based service which is primarily used for acquiring information about services that you would want to communicate with. This REST service is also known as Eureka Server. The Services that register in Eureka Server to obtain information about each other are called Eureka Clients.



How Eureka Works

Service Registration: A service registers itself with the Eureka Server upon startup.

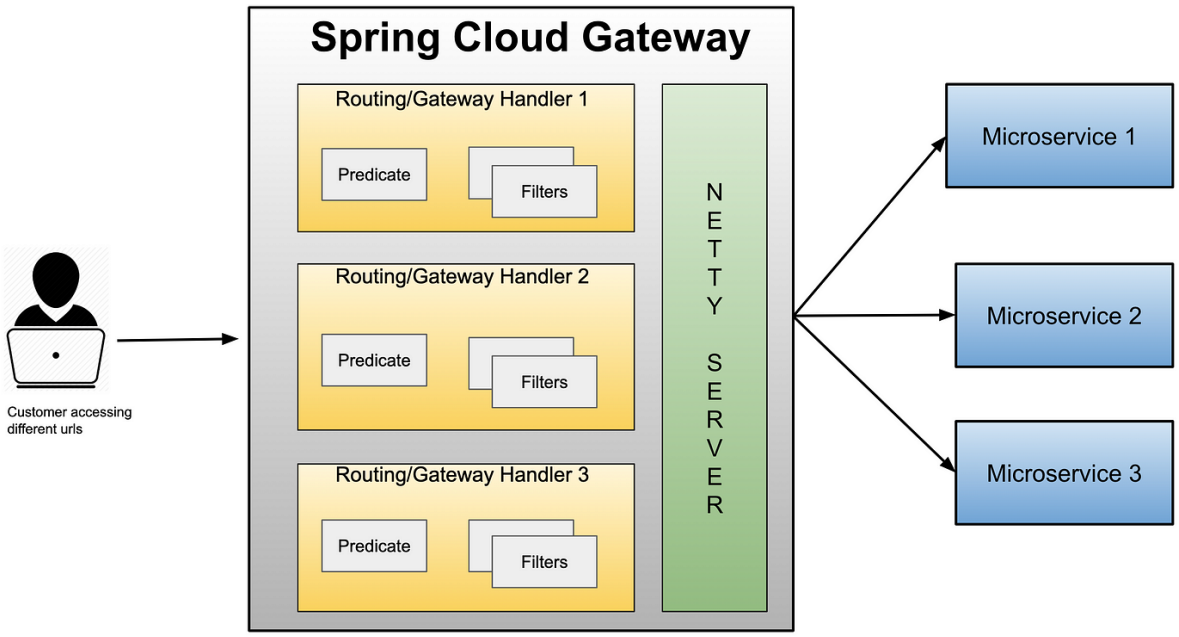
• **Heartbeat**: The service sends heartbeats periodically to renew its lease with the Eureka Server.

• **Service Discovery**: Other services can query Eureka to discover the location (IP and port) of the registered service.

• **Health Check:** Eureka performs health checks to ensure that registered services are still healthy.

• **Eviction**: If a service stops sending heartbeats and its lease expires, the Eureka Server evicts it from the registry.

#3.Spring cloud Api gateway



APIs are a common way of communication between applications. In the case of microservice architecture, there will be a number of services and the client has to know the hostnames of all underlying applications to invoke them. To simplify this communication, we prefer a component between client and server to manage all API requests called API Gateway. Additionally, we can have other features which include:

- **Security** — Authentication, authorization
- **Routing** — routing, request/response manipulation, circuit breaker
- **Observability** — metric aggregation, logging, tracing

Spring Cloud API Gateway is a powerful, flexible solution for routing and proxying requests to downstream services in a microservices architecture. It handles several important tasks like routing, filtering, authentication, and load balancing.

Spring Cloud Gateway Route

1. Route: Think of this as the destination that we want a particular request to route to. It comprises of destination URI, a condition that has to satisfy — Or in terms of technical terms, Predicates, and one or more filters.

2. Predicate: This is literally a condition to match. i.e. kind of “if” condition..if requests has something — e.g. path=blah or request header contains foo-bar etc.

Predicates with path: - Path=/api/v1/orders/**
Predicates with Method: - Method=GET
Predicates with Header: - Header=User-Agent, Mozilla/*

3. Filter: These are instances of Spring Framework WebFilter. This is where you can apply your magic of modifying request or response. There are quite a lot of out of box WebFilter that framework provides.

Config-Server_for_microservices / apiGatewayService.yml

ARONAGENT Update apiGatewayService.yml

Code | Blame | 27 lines (25 loc) · 689 Bytes | Code 55% faster with GitHub Copilot

```yaml
jwt:
  secretKey: J/2eCtv4pW7qlF6HQIuKvxU8EuhymVJZzefLq0PAXaY=

server:
  port: 8081
spring:
  cloud:
    gateway:
      routes:
        - id: orderService
          uri: lb://ORDERSERVICE
          predicates:
            - Path=/api/v1/orders/**
          filters:
            - AddRequestHeader=X-Request-Id, Rohan
            - StripPrefix=2
            - name: LoggingOrdersFilter
            # - name: Authentication
        - id: inventoryService
          uri: lb://INVENTORYSERVICE
          predicates:
            - Path=/api/v1/inventory/**
          filters:
            - AddRequestHeader=X-Request-Id, Rohan
            - StripPrefix=2
            # - name: Authentication
```

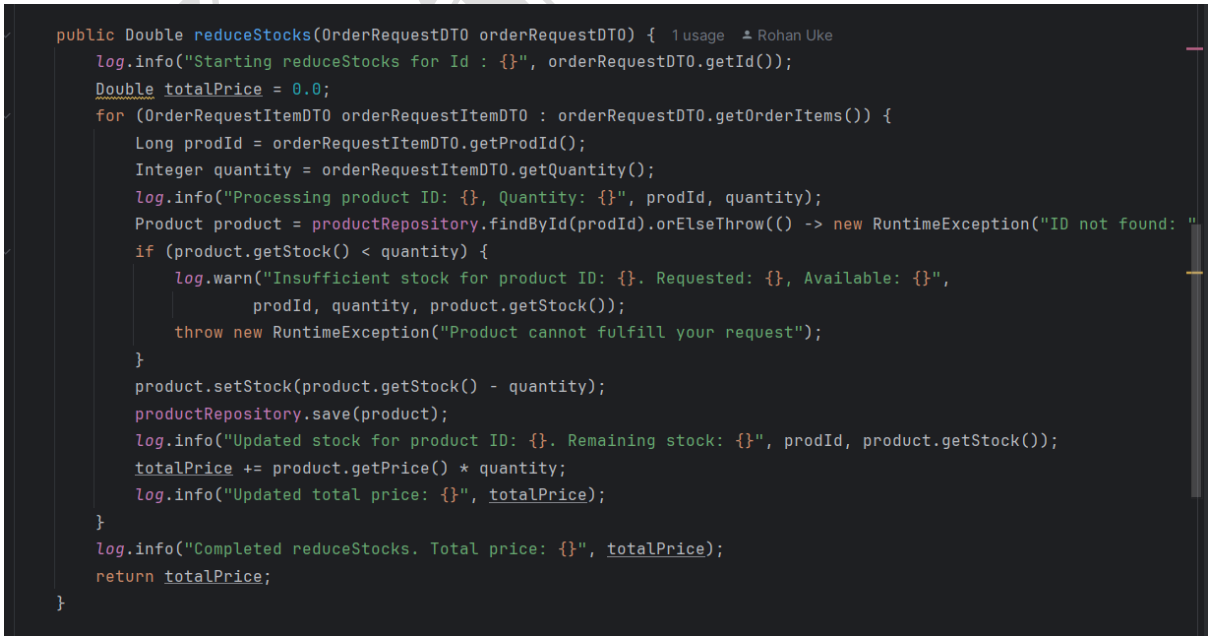
#4. Spring Cloud OpenFeign

Spring Cloud OpenFeign is a declarative HTTP client library for building RESTful microservices. It integrates seamlessly with Spring Cloud and simplifies the development of HTTP clients by allowing you to create interfaces that resemble the API of the target service. It abstracts away much of the boilerplate code typically associated with making HTTP requests, making your codebase cleaner and more maintainable

Open Feign seamlessly integrates with service discovery and load balancing provided by tools like Eureka. It allows you to refer to services by their names rather than specific host and port, which is essential in a microservices architecture.

Example

```
@FeignClient(name = "inventoryService",path ="/inventory")  2 usages  Rohan Uke
public interface InventoryFeignClient {
    @PutMapping("/products/updateStock")  1 usage  Rohan Uke
    Double reduceStock(@RequestBody OrderRequestDTO orderRequestDTO);
```



```
public Double reduceStocks(OrderRequestDTO orderRequestDTO) {  1 usage  Rohan Uke
    log.info("Starting reduceStocks for Id : {}", orderRequestDTO.getId());
    Double totalPrice = 0.0;
    for (OrderRequestItemDTO orderRequestItemDTO : orderRequestDTO.getOrderItems()) {
        Long prodId = orderRequestItemDTO.getProdId();
        Integer quantity = orderRequestItemDTO.getQuantity();
        log.info("Processing product ID: {}, Quantity: {}", prodId, quantity);
        Product product = productRepository.findById(prodId).orElseThrow(() -> new RuntimeException("ID not found: "
        if (product.getStock() < quantity) {
            log.warn("Insufficient stock for product ID: {}. Requested: {}, Available: {}",
                    prodId, quantity, product.getStock());
            throw new RuntimeException("Product cannot fulfill your request");
        }
        product.setStock(product.getStock() - quantity);
        productRepository.save(product);
        log.info("Updated stock for product ID: {}. Remaining stock: {}", prodId, product.getStock());
        totalPrice += product.getPrice() * quantity;
        log.info("Updated total price: {}", totalPrice);
    }
    log.info("Completed reduceStocks. Total price: {}", totalPrice);
    return totalPrice;
}
```

#5. Resillience4J

Resilience4j is a lightweight, standalone fault-tolerance library for Java, inspired by Netflix Hystrix but designed for Java 8 and functional programming. It is tailored for applications based on microservices architecture and provides a variety of tools to improve system resilience and stability. Resilience4j works well with frameworks such as Spring Boot and supports non-blocking and reactive programming models.

Key Features of Resilience4j:

1. Retry

Retry allows you to automatically retry a failed operation a specified number of times before giving up. This is useful when failures are transient and may succeed if attempted again after a short delay.

Use Case: Retry a remote service call that may temporarily fail due to network issues or service overload.

Configurable Options:

- Maximum retry attempts
- Wait duration between attempts
- Exceptions to include or ignore

2. Rate Limiter

The RateLimiter restricts the number of calls that can be made within a certain time window. This helps protect your services from being overwhelmed by too many requests.

Use Case: Prevent abuse or overload by limiting access to a particular API to, say, 10 calls per second.

Configurable Options:

- Number of allowed calls per time period

- Refresh period
- Timeout duration (for waiting until the next call is allowed)

3. Circuit Breaker

The CircuitBreaker monitors remote calls and opens the circuit (blocks calls) when a specified failure threshold is reached. It then allows limited calls during a “half-open” state to check if the service has recovered.

Use Case: Avoid repeatedly calling a failing downstream service and give it time to recover.

States:

- **Closed:** Normal operation; calls are allowed.
- **Open:** Calls are blocked; the service is considered unavailable.
- **Half-Open:** Limited test calls are allowed to check recovery.

Configurable Options:

- Failure rate threshold
- Wait duration in open state
- Sliding window type and size

Benefits of Using Resilience4j:

- Modular and lightweight (you can include only the modules you need)
- Built for functional programming with Java 8+ features
- Easy to integrate with Spring Boot
- Offers metrics and monitoring via Micrometer
- Works well with reactive libraries like Project Reactor

📦 Available Modules

- resilience4j-retry
- resilience4j-ratelimiter
- resilience4j-circuitbreaker
- resilience4j-timelimiter
- resilience4j-bulkhead
- resilience4j-cache
- resilience4j-micrometer

🛠️ Integration Example (Spring Boot)

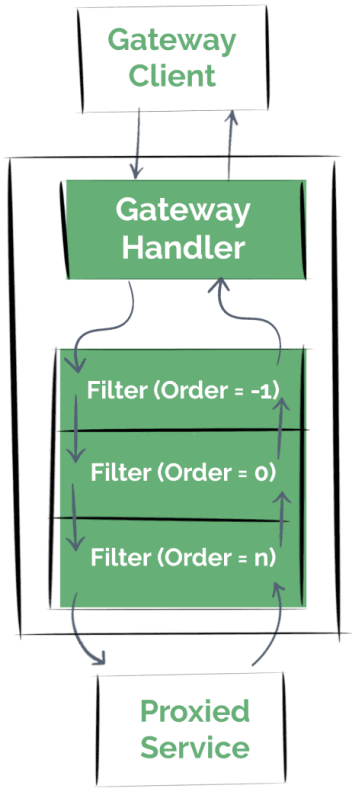
```java
@Transactional 1 usage  Rohan Uke
@CircuitBreaker(name = "inventoryCircuitBreaker",fallbackMethod = "createOrderFallback")
public OrderRequestDTO createOrder(OrderRequestDTO orderRequestDTO) {
    log.info("➡️ Starting createOrder for ID: {}", orderRequestDTO.getId());
    Double totalPrice = inventoryFeignClient.reduceStock(orderRequestDTO);
    Orders orders = modelMapper.map(orderRequestDTO, Orders.class);
    for (OrderItem item : orders.getOrderItems()) {
        item.setOrders(orders);
    }
    orders.setTotalPrice(totalPrice);
    orders.setOrderStatus(OrderStatus.CONFIRMED);
    ordersRepository.save(orders);
    log.info("✅ Finished createOrder for ID: {}, Total Price: {}", orders.getId(), orders.getTotalPrice());
    return modelMapper.map(orders, OrderRequestDTO.class);
}
```

```java
public OrderRequestDTO createOrderFallback(OrderRequestDTO orderRequestDTO, Throwable throwable) {  no usages  Rohan Uke
    log.error("Fallback occurred due to: {}", throwable.getMessage());
    return new OrderRequestDTO();
}
```

📘 Conclusion

Resilience4j is a robust and highly customizable library that empowers developers to build reliable, fault-tolerant microservices. Its minimal dependencies, functional interfaces, and rich set of modules make it an excellent choice for Java developers looking to implement resilience patterns effectively.

#6 Global And Custom Gateway Filters



API Gateway filters are used to intercept, modify, and enhance requests and responses that pass through an API Gateway. They allow you to apply common cross-cutting concerns (such as authentication, logging, rate limiting, and transformation) at a centralized entry point before routing requests to microservices. There are two types of filters:

1. **Global Filters**

2. **Route specific Filters**

1.Global Filter

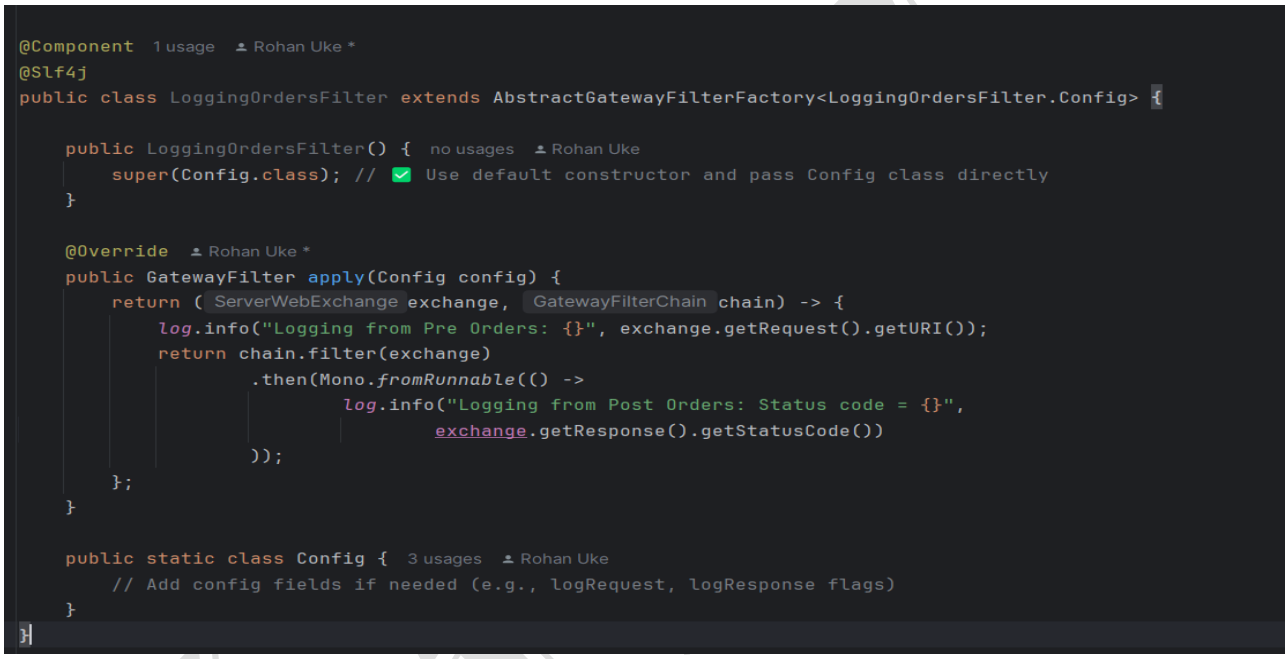
All we have to do to create a custom global filter is to implement the Spring Cloud Gateway GlobalFilter interface, and add itto the context as a bean.

```java
import org.springframework.cloud.gateway.filter.GatewayFilterChain;
import org.springframework.cloud.gateway.filter.GlobalFilter;
import org.springframework.core.Ordered;
import org.springframework.stereotype.Component;
import org.springframework.web.server.ServerWebExchange;
import reactor.core.publisher.Mono;

@Component  no usages  Rohan Uke
@Slf4j
public class GlobalLoginGatewayFilter implements GlobalFilter, Ordered {
    @Override  Rohan Uke
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {

       log.info("Logging from Pre Global {}",exchange.getRequest().getURI());
        return chain.filter(exchange).then(Mono.fromRunnable(()->{
            log.info("Logging from Post Global {}",exchange.getResponse().getStatusCode());
        }));
    }

    @Override  Rohan Uke
    public int getOrder() {
        return 5;
    }
}
```

#2.Route Specific Filter

Global filters are quite useful, but we often need to execute fine-grained custom Gateway filter operations that apply to only some routes.

Route specific Filters. To implement a GatewayFilter, we'll have to extend from the AbstractGatewayFilterFactory class provided by Spring Cloud Gateway.

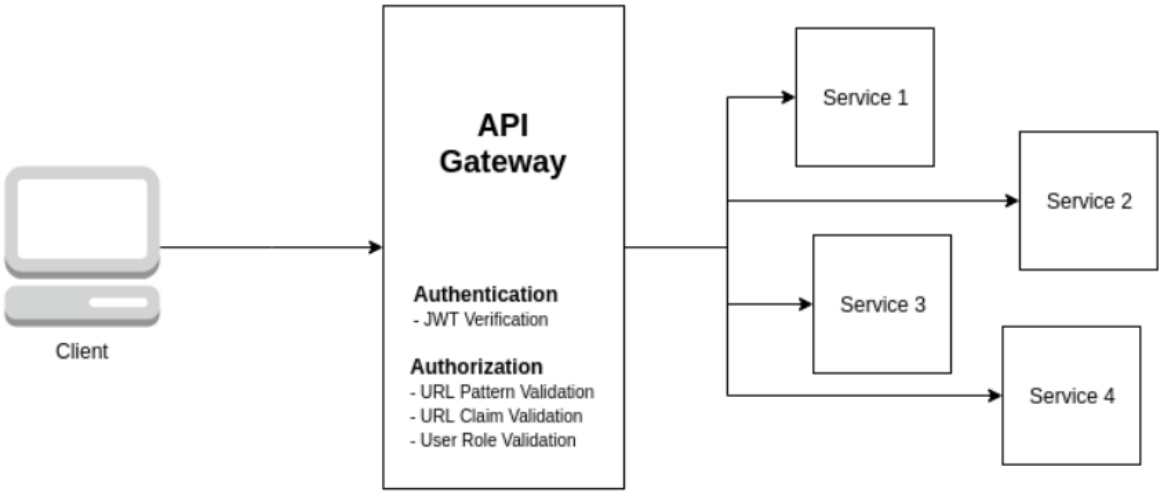


```
@Component  1 usage   Rohan Uke *
@Slf4j
public class LoggingOrdersFilter extends AbstractGatewayFilterFactory<LoggingOrdersFilter.Config> {

    public LoggingOrdersFilter() {  no usages   Rohan Uke
        super(Config.class); //  Use default constructor and pass Config class directly
    }

    @Override   Rohan Uke *
    public GatewayFilter apply(Config config) {
        return ( ServerWebExchange exchange,  GatewayFilterChain chain) -> {
            log.info("Logging from Pre Orders: {}", exchange.getRequest().getURI());
            return chain.filter(exchange)
                    .then(Mono.fromRunnable(() ->
                            log.info("Logging from Post Orders: Status code = {}",
                                    exchange.getResponse().getStatusCode())
                    ));
        };
    }

    public static class Config {  3 usages   Rohan Uke
        // Add config fields if needed (e.g., logRequest, logResponse flags)
    }
}
```

#7.Authentcation in centralized api gateway Service

The API Gateway acts as a central authentication point for all incoming requests. Clients authenticate once with the API Gateway, which validates the credentials (could be a JWT, OAuth2, or any other method). The API Gateway forwards the request to the respective microservice with the user's credentials or token.

API Gateway authentication is the process of verifying the identity of clients accessing an API through an API Gateway, protecting it from unauthorized access and potential security threats.



Mutate the Request

You can mutate the request (e.g., add headers, modify path, or change query parameters) before it is sent to the downstream service. This is typically done in pre-filters.

```
@Override  Rohan Uke
public GatewayFilter apply(Config config) {
    return ( ServerWebExchange exchange,  GatewayFilterChain chain) -> {

        String authenticationHeader=exchange.getRequest().getHeaders()
                .getFirst( headerName: "Authorization");
        if(authenticationHeader==null) {
            exchange.getResponse().setStatusCode(HttpStatus.UNAUTHORIZED);
            return exchange.getResponse().setComplete();
        }
        String token=authenticationHeader.split( regex: "Bearer ")[1];

        Long userId=jwtService.getUserIdFromToken(token);

        // Spring Boot 3.4.1
        ServerHttpRequest mutatedRequest=exchange.getRequest().mutate()
                .header( headerName: "X_User-Id",userId.toString())
                .build();


        return chain.filter(exchange.mutate().request(mutatedRequest).build());
    };
}
```

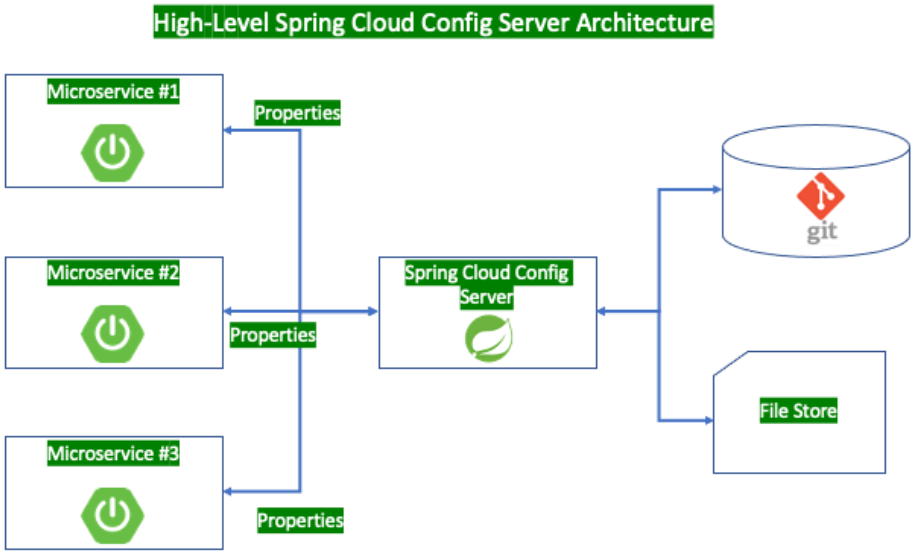
Mutate the Response

You can modify the response (e.g., add headers, modify the body) before it is sent to the client. This is typically done in post-filters.

```
@Override  Rohan Uke *
public GatewayFilter apply(Config config) {
    return ( ServerWebExchange exchange,  GatewayFilterChain chain) -> {
        log.info("Logging from Pre Orders: {}", exchange.getRequest().getURI());
        return chain.filter(exchange)
                .then(Mono.fromRunnable(() ->
                        log.info("Logging from Post Orders: Status code = {}",
                                exchange.getResponse().getStatusCode())
                ));
    };
}
```

#8 Centralized Config Server

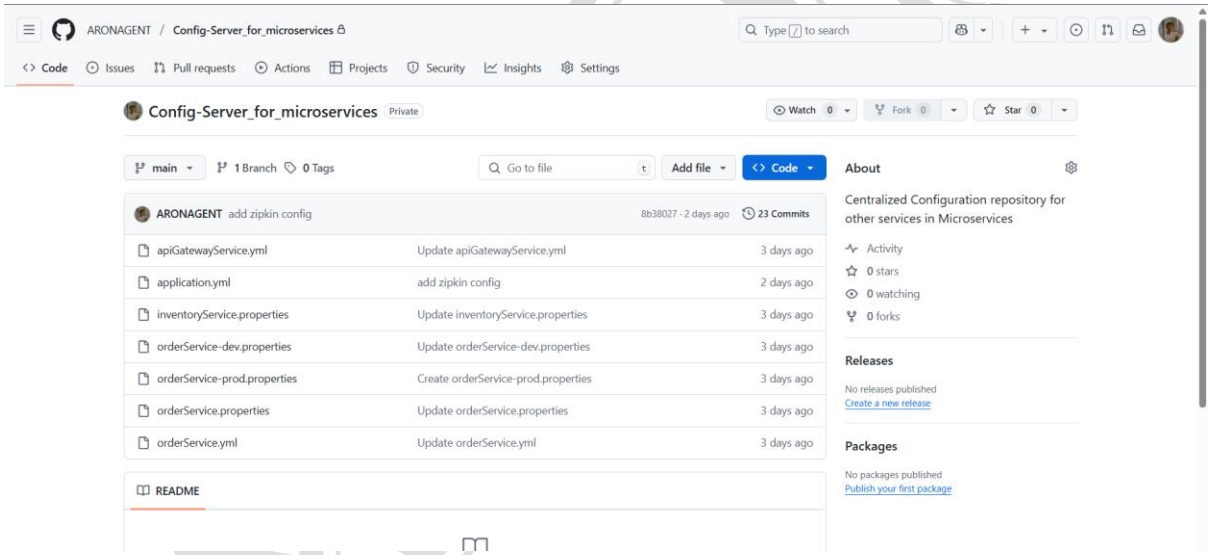
Config Server is a dedicated service that acts as a centralized repository, allowing all microservices to retrieve their configurations dynamically at runtime. Config Server can manage different environments (e.g., dev, test,prod) and profiles. This means that each environment can have its own set of properties (e.g., different databases or API keys).



Configuration of Config Server

```
spring.application.name= config-server

spring.cloud.config.server.git.uri= https://github.com/ARONAGENT/Config-Server_for_microservices
spring.cloud.config.server.git.username= ARONAGENT
spring.cloud.config.server.git.password= ${access_token}
spring.cloud.config.server.git.default-label= main

sever.port=8888



eureka.client.service-url.defaultZone= http://localhost:8761/eureka
```

All configs manage by This Repository



#9 What Are Spring Profiles?

Spring Profiles provide a way to segregate parts of your application configuration and make it only available in certain environments. It is an essential feature for managing environment-specific configurations like development, testing, staging, and production.

🌐 Why Use Spring Profiles?

Different environments require different configurations. For example:

- In **development**, you might use an in-memory database.
- In **production**, you'll use a secure, cloud-hosted database.

Spring Profiles allow you to define and activate configurations for each environment without changing your code.

📁 Defining Profiles

You define profiles in your configuration files using spring.profiles:

Example - application-dev.properties

spring.datasource.url=jdbc:h2:mem:testdb

logging.level.org.springframework=DEBUG

Example - application-prod.properties

spring.datasource.url=jdbc:mysql://prod-db-server:3306/mydb

logging.level.org.springframework=ERROR

🔄 Activating a Profile

You can activate a profile in different ways:

1. application.properties

spring.profiles.active=dev

📚 Use Case

In a microservices system:

- **dev profile** can log all incoming requests with full stack traces.
- **test profile** can use mock databases and services.
- **prod profile** will be optimized for performance and use secure settings.

📖 Conclusion

Spring Profiles offer a clean and powerful way to separate configuration concerns for different environments. They are especially useful in CI/CD pipelines and microservices deployment, allowing developers to manage properties smartly and consistently without modifying the code.

#10 .Refresh Config without Restart using Centralized Config

@RefreshScope is a Spring Cloud annotation used to enable dynamic refreshing of Spring-managed beans, allowing certain configurations in an application to be updated at runtime without the need to restart the application. It is commonly used in conjunction with Spring Cloud Config, which provides externalized configuration management for distributed systems.

Key features are:

- Dynamic Bean Refresh
- Avoiding restarts
- Actuator support for refreshing

```yaml
# actuator config
management:
  endpoints:
    web:
      exposure:
        include: "refresh"
```

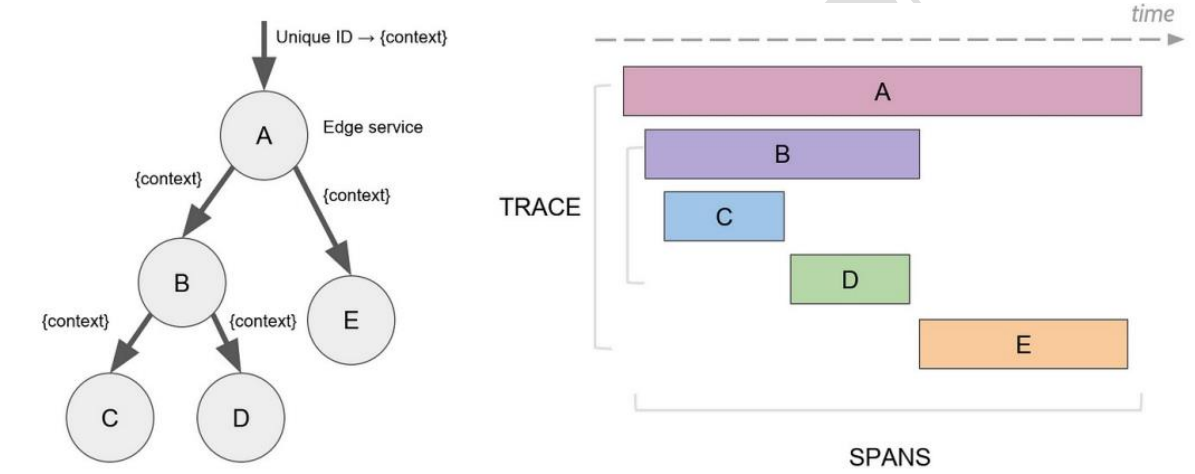
Enable Actuator Endpoints

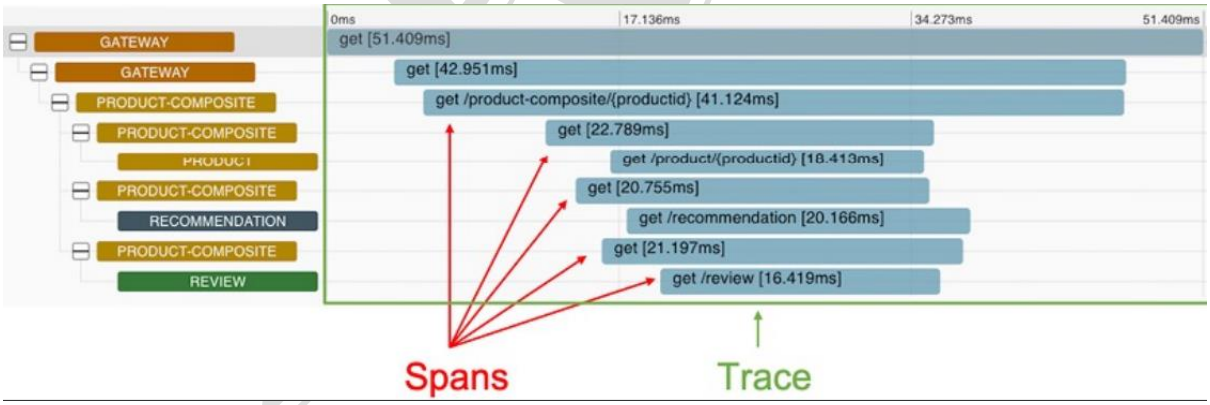
The /refresh endpoint in Spring Boot, provided by Spring Cloud Actuator, is used to dynamically refresh the configuration properties of a running application without restarting it. For the POST /actuator/refresh to work we need to enable the refresh actuator endpoint.

#11.Zipkin and Micrometer

Distributed Tracing

Distributed tracing is a technique that tracks requests as they move through a distributed system, such as a microservices environment or cloud native architecture. It helps developers understand how requests are handled across multiple applications, services, and databases.





Micrometer

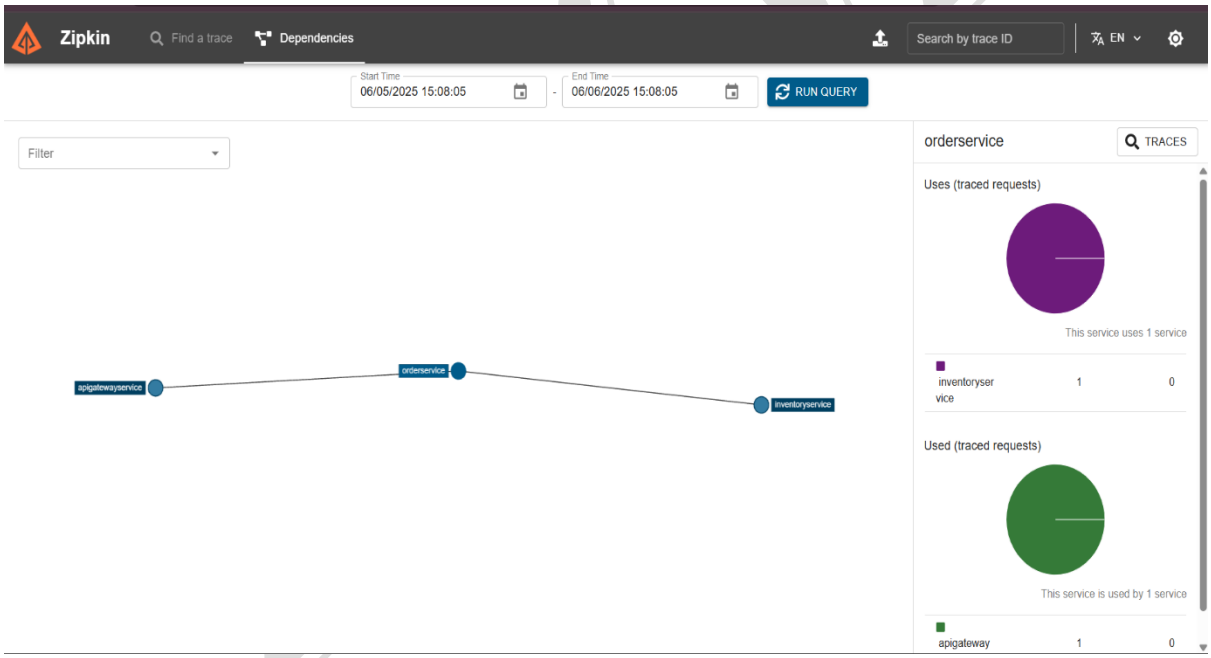
Micrometer is a tool that helps developers keep track of how their microservices are performing. It provides a simple façade over the instrumentation clients for the most popular monitoring systems. It is a vendor-neutral application metrics façade.

Application metrics recorded by Micrometer are intended to be used to observe, alert, and react to the current/recent operational state of your environment**. It can be used as metrics Façade for Amazon Cloud Watch, Elastic, Prometheus and Zipkin.**

Zipkin

Zipkin is a tool that helps developers trace requests across different parts of a distributed system. Think of it as a way to see the path a request takes as it moves between different microservices**. This can be really useful for understanding how long it takes for requests to be processed, where bottlenecks are, and what might be causing problems.**

Zipkin Dependencies Flow

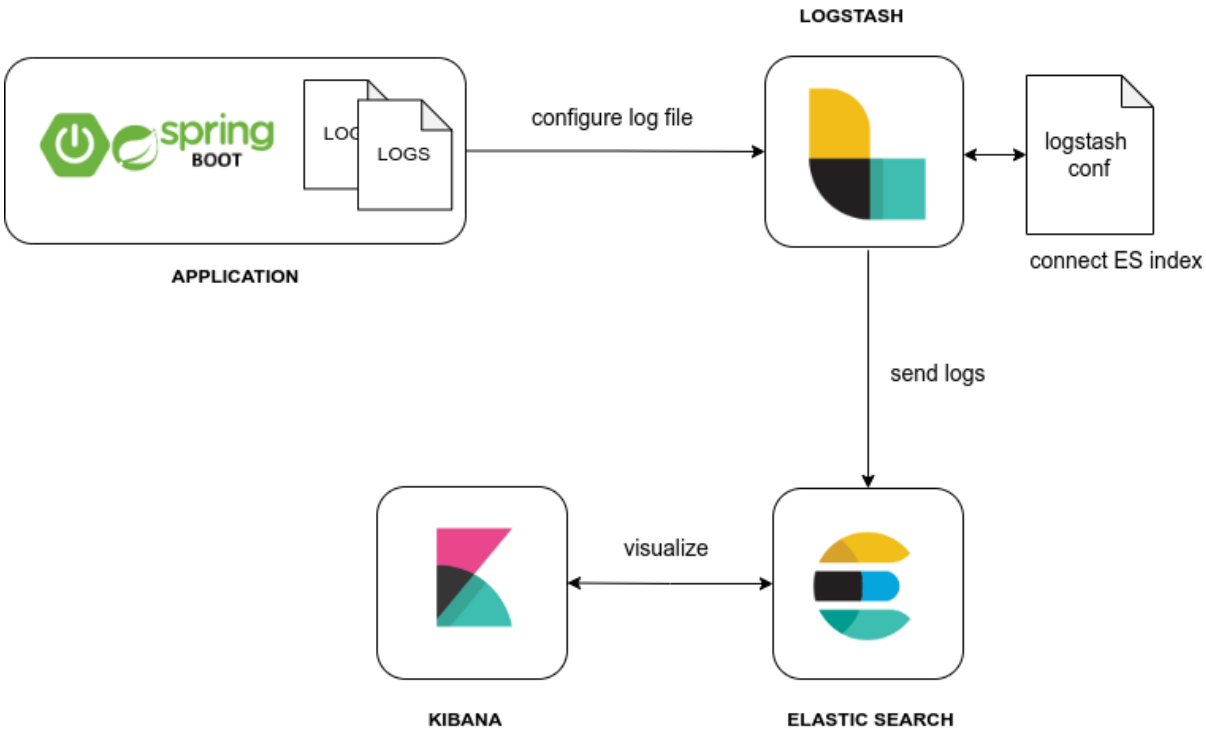


#12.ELK stack

Why Centralized Logging

While Zipkin is an excellent tool for distributed tracing and observing request paths through microservices, it doesn't offer deep log analysis. **ELK Stack complements tracing tools like Zipkin by offering:**

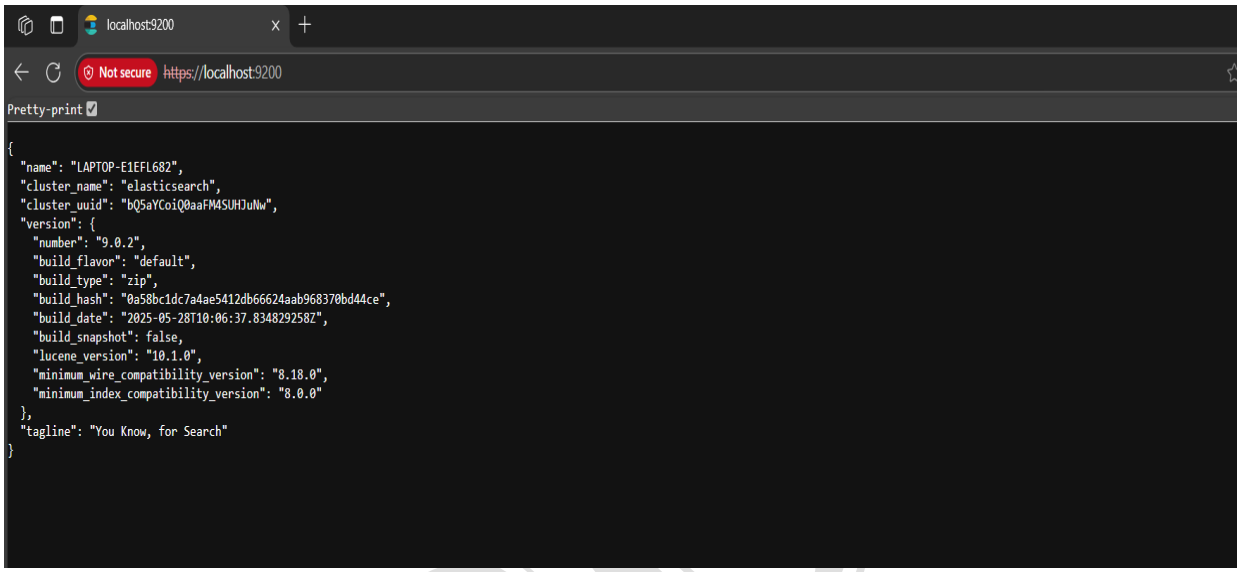
- **Deep log analysis and insights across microservices**.
- Long-term storage and powerful search capabilities.
- Real-time monitoring and alerting.
- Centralized view of logs for easier debugging and troubleshooting.



ELK is a collection of three open-source applications –

Elasticsearch, Logstash, and Kibana from Elastic that accepts data from any source or format, on which you can then perform search, analysis, and visualize that data.

1. Elasticsearch - Elasticsearch stores and indexes the data. It is a NoSQL database based on Lucene's open-source search engine. Since Elasticsearch is developed using Java, therefore, it can run on different platforms. One particular aspect where it excels is indexing streams of data such as logs.



2. Logstash — Logstash is a tool that integrates with a variety of deployments. It is used to collect, parse, transform, and buffer data from a variety of sources. The data collected by Logstash can be shipped to one or more targets like Elasticsearch.

How Does the Logstash Work

1. When new log entries are added to the specified files, Logstash reads these entries and converts them into structured events. Each line or entry in the log file is treated as a separate event.

2. Once the events are processed, Logstash sends them to the configured outputs. In this case, it sends the events to both the console (for debugging) and to Elasticsearch. The logs are sent in near real-time, allowing for timely analysis and visualization.

3. Elasticsearch receives the log events and indexes them according to the specified index pattern. This makes the logs searchable and allows for data analysis using tools like Kibana.

Configure Logstash

1. Configure Applications to send their logs to log file. Create a logbackspring.xml to generate logs.

2. Create a configuration file for Logstash – logstash.conf and save this file in the logstash installation folder.

3. Edit this file to include the input and output locations. Here the input will be the location of the logs, and the output will be the ElasticSearch index.

4. Save this file and run

logstash –f logstash.conf

3. Kibana - Kibana acts as an analytics and visualization layer on top of Elasticsearch. Kibana can be used to search, view, and interpret the data stored in Elasticsearch.