Spring Cloud Config

Objectives

After completing this lesson, you should be able to do the following

- Introduce Spring Cloud Config
- Understand the requirements for connecting and mapping the documents

Spring Cloud Config

- Spring Cloud Config provides server and client-side support for externalized configuration in a distributed system.
- With the Config Server you have a central place to manage external properties for applications across all environments.
- The concepts on both client and server map identically to the Spring Environment and PropertySource abstractions, so they fit very well with Spring applications, but can be used with any application running in any language.

Spring Cloud Config

- As an application moves through the deployment pipeline from dev to test and into production you can manage the configuration between those environments and be certain that applications have everything they need to run when they migrate.
- The default implementation of the server storage backend uses git so it easily supports labelled versions of configuration environments, as well as being accessible to a wide range of tooling for managing the content.
- It is easy to add alternative implementations and plug them in with Spring configuration.

Spring Cloud Config Features

- Spring Cloud Config Server features:
  - HTTP, resource-based API for external configuration (name-value pairs, or equivalent YAML content)
  - Encrypt and decrypt property values (symmetric or asymmetric)
  - Embeddable easily in a Spring Boot application using @EnableConfigServer

- Config Client features (for Spring applications):
  - Bind to the Config Server and initialize Spring Environment with remote property sources
  - Encrypt and decrypt property values (symmetric or asymmetric)

Spring Cloud Config – why configuration management is important

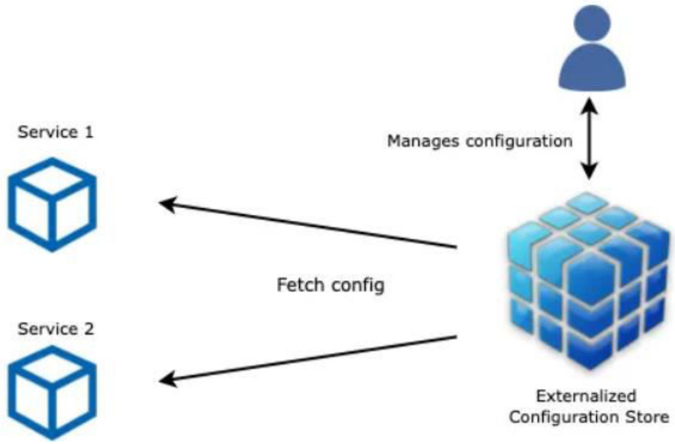
- Spring Cloud applications, especially those built on microservices architecture, rely heavily on configuration management for a number of reasons:
  - Centralized Configuration
  - Environmental Differences
  - Dynamic Updates
  - Version Control
  - Microservice Isolation

Challenges of Configuration management in microservices

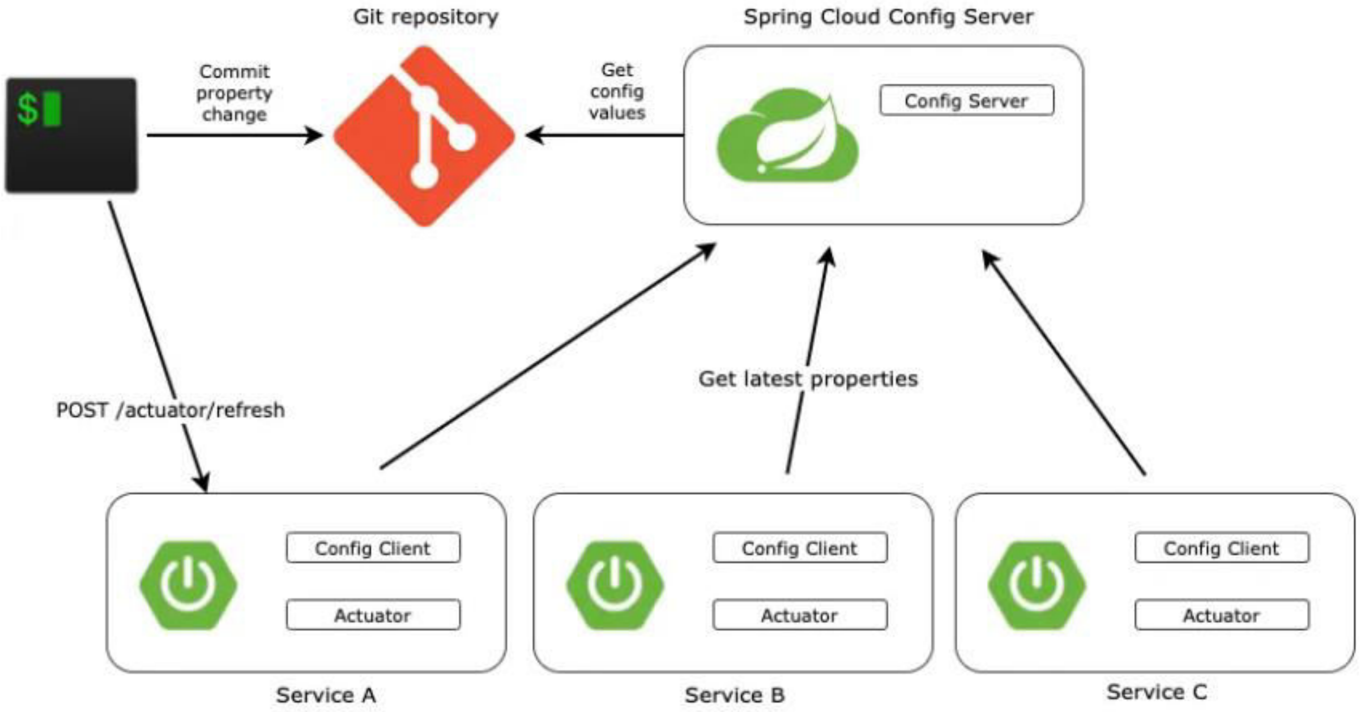
- Even though configuration management is crucial for microservices, it comes with its own set of challenges:
  - Increased Complexity
  - Versioning and Consistency
  - Security Concerns
  - Secrecy Management
  - Distributed System Issues

Decouple Configuration from Application Code with Spring Cloud Config

- Centralized configuration is a pattern where the configuration for all services is managed in a central repository.
- Decoupling configuration from services helps with easier organization and management of settings. This separation enables independent changes to configurations without the need to redeploy services.



Centralized Configuration Service



Understanding Externalized Configuration

- Externalized configuration is a pattern in software design where configuration settings are separated from the application code.
- This pattern becomes increasingly essential as applications grow in complexity, requiring more configuration settings for different environments, instances, and contexts.

Understanding Externalized Configuration

- Externalized configuration is a pattern in software design where configuration settings are separated from the application code.

- This pattern becomes increasingly essential as applications grow in complexity, requiring more configuration settings for different environments, instances, and contexts.

- In traditional application development, configurations are often bundled within the application, making it difficult to adapt the application to various environments (development, testing, production) without modifying the code or rebuilding the entire application.

Dynamic Configuration

- By keeping the configuration separate and in an external repository, you can change the configuration of your application dynamically at runtime. When the application starts up, it queries the Spring Cloud Config Server for its configuration.
- The Config Server then pulls the latest configuration from the external repository and serves it to the application.

Separation of Configuration

- The separation of configuration from the application code allows developers and operations teams to manage and modify configurations independently, providing more flexibility and control over the application's behavior in different environments.

- This approach also streamlines the process of migrating applications through different environments during the development lifecycle.

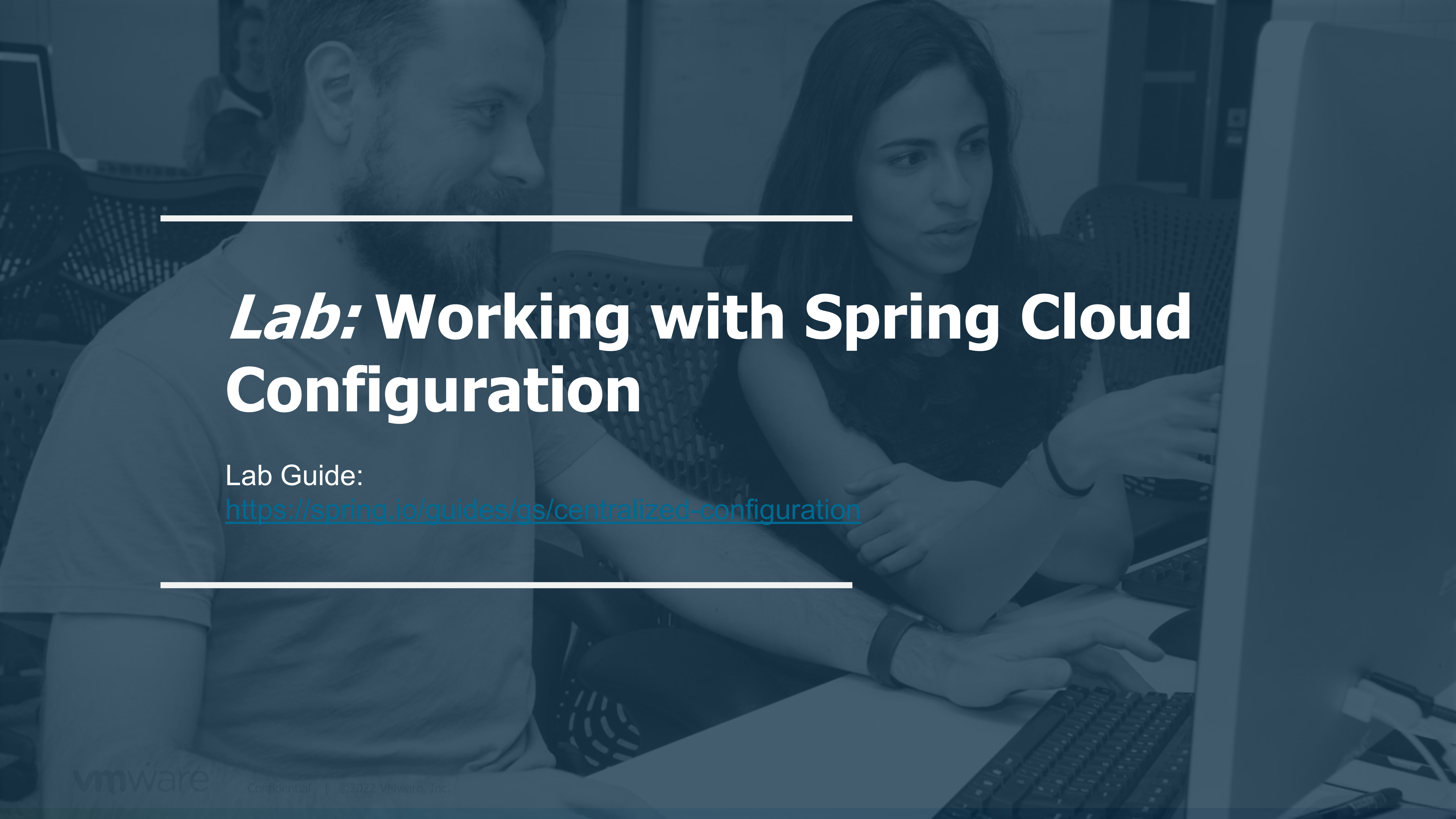
- Configuration for each environment can be stored in the external repository, and the application can pull the appropriate configuration based on its current environment at startup.

Utilizing @RefreshScope for Dynamic Updates

- In a microservices environment, it's crucial to ensure that configuration changes do not require a complete system restart for the changes to take effect.

- @RefreshScope in Spring Cloud provides a mechanism for this. It allows beans in the Spring ApplicationContext to be re-initialized on demand to pick up refreshed or updated configuration values from an external source like a Git repository managed by Spring Cloud Config Server.

- When you annotate a Spring @Bean with @RefreshScope, the bean gets special treatment. The @RefreshScope annotation marks the bean to be refreshed on a configuration change.



Lab: Working with Spring Cloud Configuration

Lab Guide:
https://spring.io/guides/gs/centralized-configuration

Spring Cloud Eureka

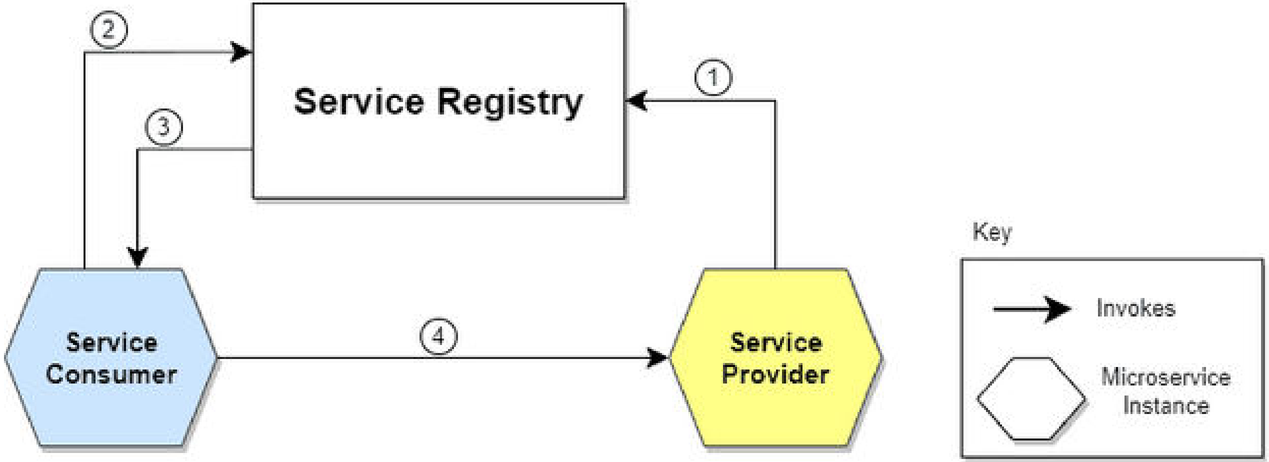
Service discovery

Service discovery in Microservices

- A microservices-based application typically runs in virtualized or containerized environments. The number of instances of a service and its locations changes dynamically.
- It is need to know where these instances are and their names to allow requests to arrive at the target microservice.
- The Service Discovery mechanism helps us know where each instance is located. In this way, a Service Discovery component acts as a registry in which the addresses of all instances are tracked.
- The instances have dynamically assigned network paths. Consequently, if a client wants to make a request to a service, it must use a Service Discovery mechanism.

How Does Service Discovery Works?

- Service Discovery handles things in two parts. First, it provides a mechanism for an instance to register and say, “I’m here!” Second, it provides a way to find the service once it has registered.



Spring Netflix Eureka

- Spring Cloud Netflix provides Netflix OSS integrations for Spring Boot apps through autoconfiguration and binding to the Spring Environment and other Spring programming model idioms.
- With a few simple annotations, you can quickly enable and configure the common patterns inside your application and build large distributed systems with battle-tested Netflix components.
- The patterns provided include Service Discovery (Eureka).

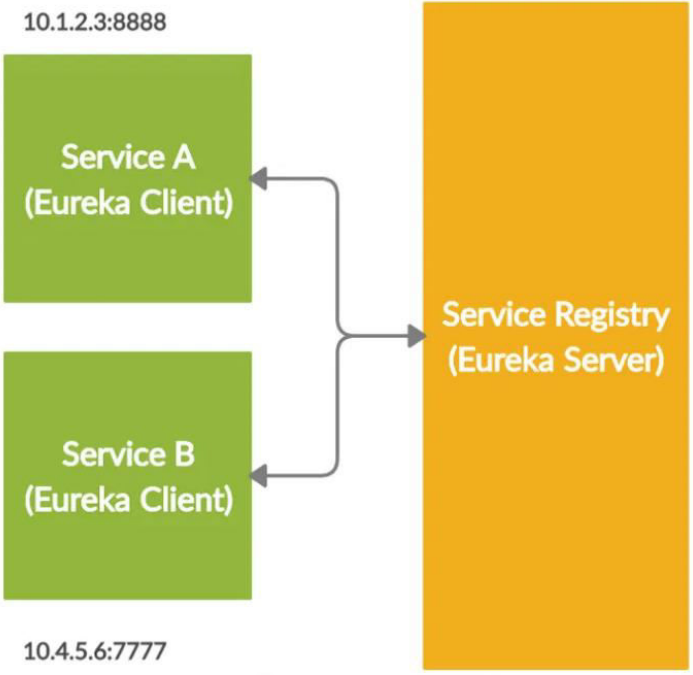
Spring Netflix Eureka - Features

Spring Cloud Netflix features:

- Service Discovery: Eureka instances can be registered and clients can discover the instances using Spring-managed beans
- Service Discovery: an embedded Eureka server can be created with declarative Java configuration

Spring Netflix 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.



Spring Netflix Eureka

Eureka mainly consists of main components, let’s see what they are:

1. Eureka Server: It is an application that contains information about all client service applications. Each microservice is registered with the Eureka server and Eureka knows all the client applications running on each port and IP address. Eureka Server is also known as Discovery Server.

2. Eureka Client: It’s the actual microservice and it registers with the Eureka Server, so if any other microservice wants the Eureka Client’s address then they’ll contact the Eureka Server.

Spring Netflix Eureka Server

Implementing a Eureka Server for service registry is as below:

- Adding spring-cloud-starter-netflix-eureka-server to the dependencies

- Enabling the Eureka Server in a @SpringBootApplication by annotating it with @EnableEurekaServer

- Configuring some properties

Spring Netflix Eureka Client

- For a @SpringBootApplication to be discovery-aware, we have to include a Spring Discovery Client (for example, spring-cloud-starter-netflix-eureka-client) into our classpath.

- Then we need to annotate a @Configuration with either @EnableDiscoveryClient or @EnableEurekaClient. Note that this annotation is optional if we have the spring-cloud-starter-netflix-eureka-client dependency on the classpath.

- The latter tells Spring Boot to use Spring Netflix Eureka for service discovery explicitly.

Lab: Working with Spring Service Registration and Discovery using Netflix Eureka

Lab Guide:
https://spring.io/guides/gs/service-registration-and-discovery

Spring Cloud Circuit Breaker – Resilience4J

Resilience4J

- Resilience4j is a lightweight fault tolerance library designed for functional programming.
- Resilience4j provides higher-order functions (decorators) to enhance any functional interface, lambda expression or method reference with a Circuit Breaker, Rate Limiter, Retry or Bulkhead.
- It can stack more than one decorator on any functional interface, lambda expression or method reference.

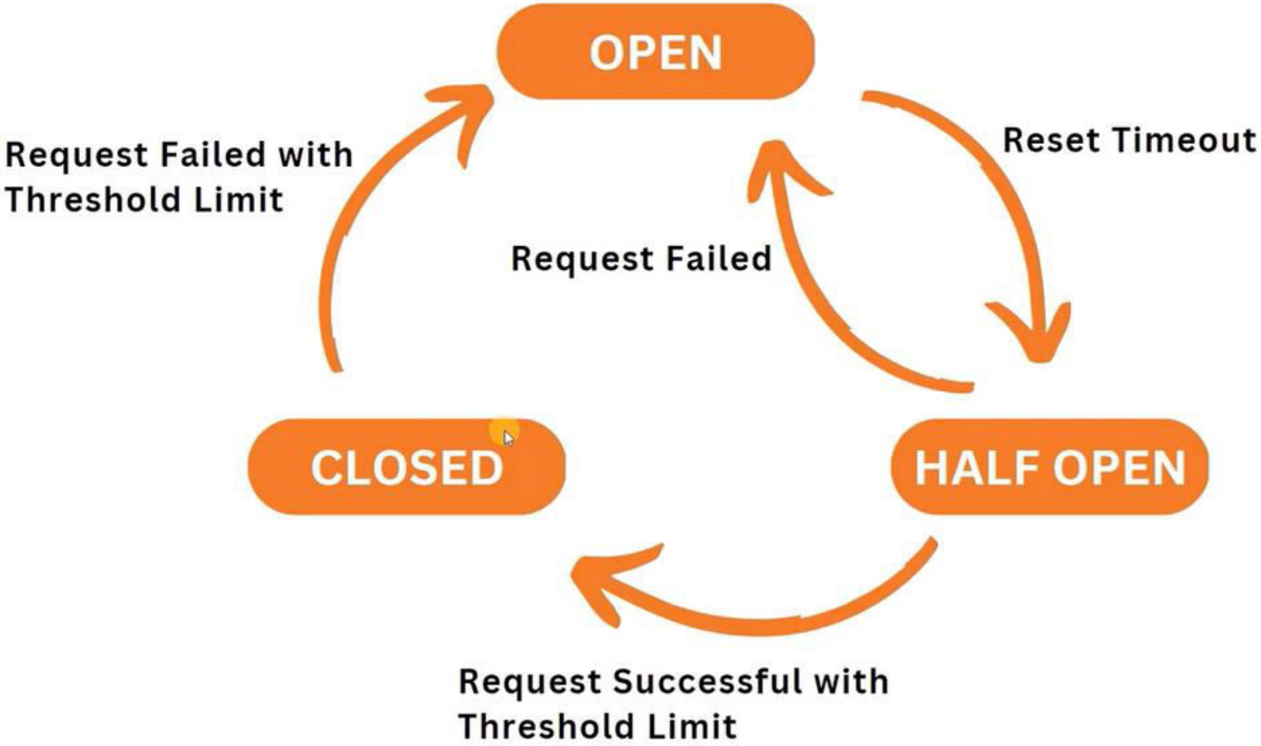
Resilience4J – Features:

- Circuit Breaker
- Retry
- Bulkhead
- RateLimiting
- TimeLimiting

Resilience4J – Circuit Breaker

- Circuit Breaker is a resilience pattern that helps to prevent cascading failures in distributed systems. It acts as a safety mechanism that isolates failing components to prevent further errors from propagating.
- The CircuitBreaker uses a sliding window to store and aggregate the outcome of calls. It can choose between a count-based sliding window and a time-based sliding window.
- The count-based sliding window aggregrates the outcome of the last N calls. The time-based sliding window aggregrates the outcome of the calls of the last N seconds.

Spring Cloud Circuit Breaker


OPEN
Reset Timeout
Request Failed with Threshold Limit
Request Failed
CLOSED
HALF OPEN
Request Successful with Threshold Limit

Resilience4J – Circuit Breaker

- Open State:
  - When a component fails a certain number of times within a specified window, the circuit breaker transitions to the open state.
  - In this state, all subsequent requests are immediately rejected without attempting to call the component.

- Closed State:
  - If the component successfully recovers from the failure, the circuit breaker transitions to the closed state.
  - In this state, requests are allowed to proceed normally.

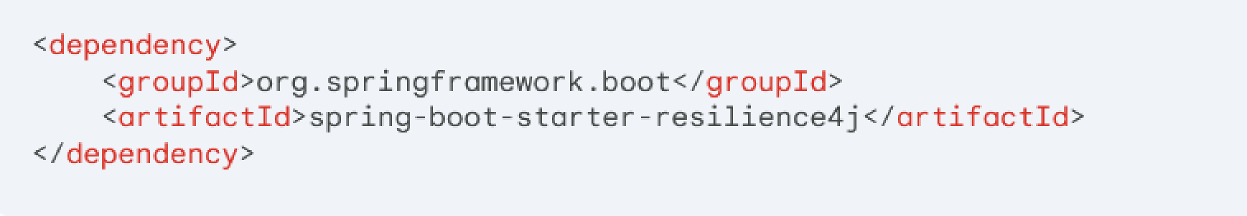
Resilience4J – Circuit Breaker

- Half-Open State:
  - After a specified wait duration, the circuit breaker enters the half-open state. In this state, a limited number of requests are allowed to pass through.
  - If these requests are successful, the circuit breaker transitions to the closed state. Otherwise, it remains in the open state.

- Fallback Mechanism:
  - It allows to configure a fallback mechanism to be executed when the circuit breaker is open.
  - This allows you to provide a graceful degradation of service or return a default response.

Resilience4J – Circuit Breaker

- Properties and Configuration:



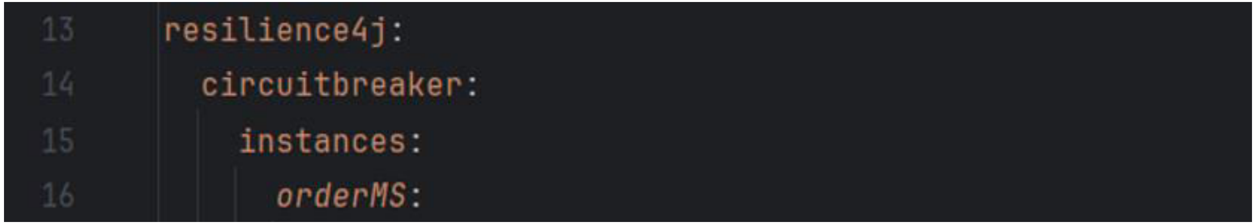
```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-resilience4j</artifactId>
</dependency>
```

- Configuration

  In Controller :

```
@CircuitBreaker(name = "orderMS", fallbackMethod =
"fallbackMethod")
```

  In Properties:



```
resilience4j:
  circuitbreaker:
    instances:
      orderMS:
```

Resilience4J – Circuit Breaker

- Properties and Configuration:

Parameter	Description	Default Value
name	Unique identifier for the circuit breaker.	default
failureRateThreshold	Percentage of failed calls within a sliding window that triggers the open state.	50%
waitDuration	Duration the circuit breaker remains open after it trips.	60 seconds
slidingWindowSize	Size of the sliding window used to calculate the failure rate.	10
recordFailurePredicate	Predicate used to determine if a call should be considered a failure.	Default predicate based on exceptions

Resilience4J – Circuit Breaker

- Properties and Configuration:

Parameter	Description	Default Value
slowCallDurationThreshold	Threshold for considering a call as slow.	null (not enabled by default)
automaticTransitionToHalfOpen	Whether the circuit breaker should automatically transition to the half-open state after the wait duration.	TRUE
halfOpenWindow	Size of the half-open window.	10
callCombineFunction	Function to combine multiple calls into a single result.	null (no combination by default)
slowCallRateThreshold	Threshold for the rate of slow calls.	null (not enabled by default)

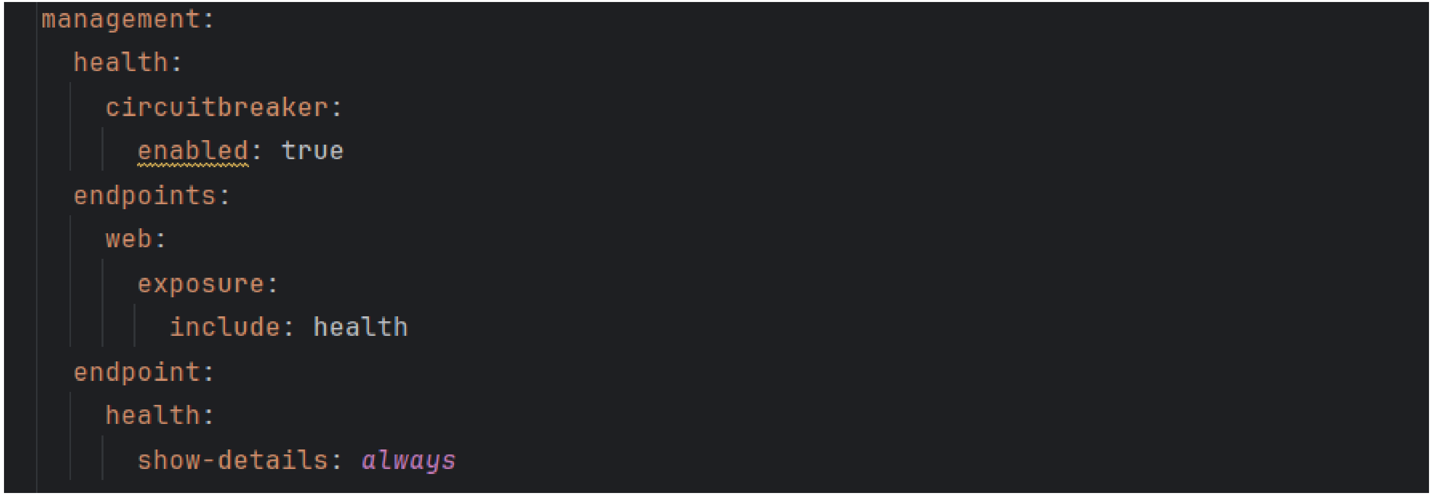
Resilience4J – Circuit Breaker

- Collecting metrics in Resilience4J Circuit Breaker

Parameter	Description
success	Number of successful calls.
failure	Number of failed calls.
slowCall	Number of calls that exceeded the slow call duration threshold.
notPermitted	Number of calls that were not permitted due to the circuit breaker being open.
error	Number of calls that resulted in an error.

Resilience4J – Circuit Breaker

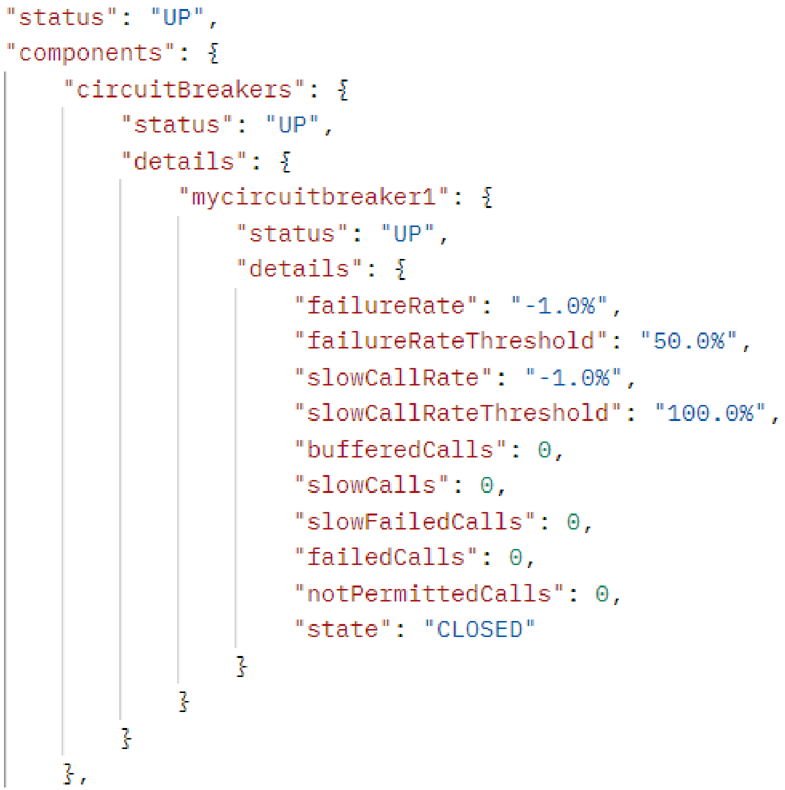
- Collecting metrics in Resilience4J Circuit Breaker
  - Configuring metrics in Resilience4J Circuit Breaker



```yaml
management:
  health:
    circuitbreaker:
      enabled: true
  endpoints:
    web:
      exposure:
        include: health
  endpoint:
    health:
      show-details: always
```

Resilience4J – Circuit Breaker

- Collecting metrics in Resilience4J Circuit Breaker



```
"status": "UP",
"components": {
    "circuitBreakers": {
        "status": "UP",
        "details": {
            "mycircuitbreaker1": {
                "status": "UP",
                "details": {
                    "failureRate": "-1.0%",
                    "failureRateThreshold": "50.0%",
                    "slowCallRate": "-1.0%",
                    "slowCallRateThreshold": "100.0%",
                    "bufferedCalls": 0,
                    "slowCalls": 0,
                    "slowFailedCalls": 0,
                    "failedCalls": 0,
                    "notPermittedCalls": 0,
                    "state": "CLOSED"
                }
            }
        }
    },
```

Resilience4J – Bulkhead

- **Bulkhead** is a resilience pattern that isolates groups of resources to prevent failures from affecting the entire application.
- In Resilience4J, it's used to limit the number of concurrent calls to a specific resource or group of resources.

Parameter	Description	Default Value
maxConcurrentCalls	Maximum number of concurrent calls allowed.	10
maxWaitDuration	Maximum duration a call can wait for a slot before rejecting.	Duration.ZERO (no waiting)
fallbackStrategy	Strategy to use when the bulkhead is full.	REJECT (reject the call)

Resilience4J – RateLimiting

- **Resilience4J Ratelimiter** is a powerful tool for controlling the rate of incoming requests to a service or resource.
- It helps prevent overload and ensures that resources are not overwhelmed by excessive traffic.

Parameter	**Description**	**Default Value**
name	Unique identifier for the ratelimiter.	default
limit	Maximum number of requests allowed within the specified time window.	10
timeout	Time window for rate limiting.	Duration.ofSeconds (1)
limitRefreshPeriod	Period at which the limit is refreshed.	Duration.ofSeconds (1)
rejectionStrategy	Strategy to use when the rate limit is exceeded.	REJECT

Resilience4J – Retry

- The Retry pattern is a resilience mechanism that automatically retries a failed operation a specified number of times before giving up.

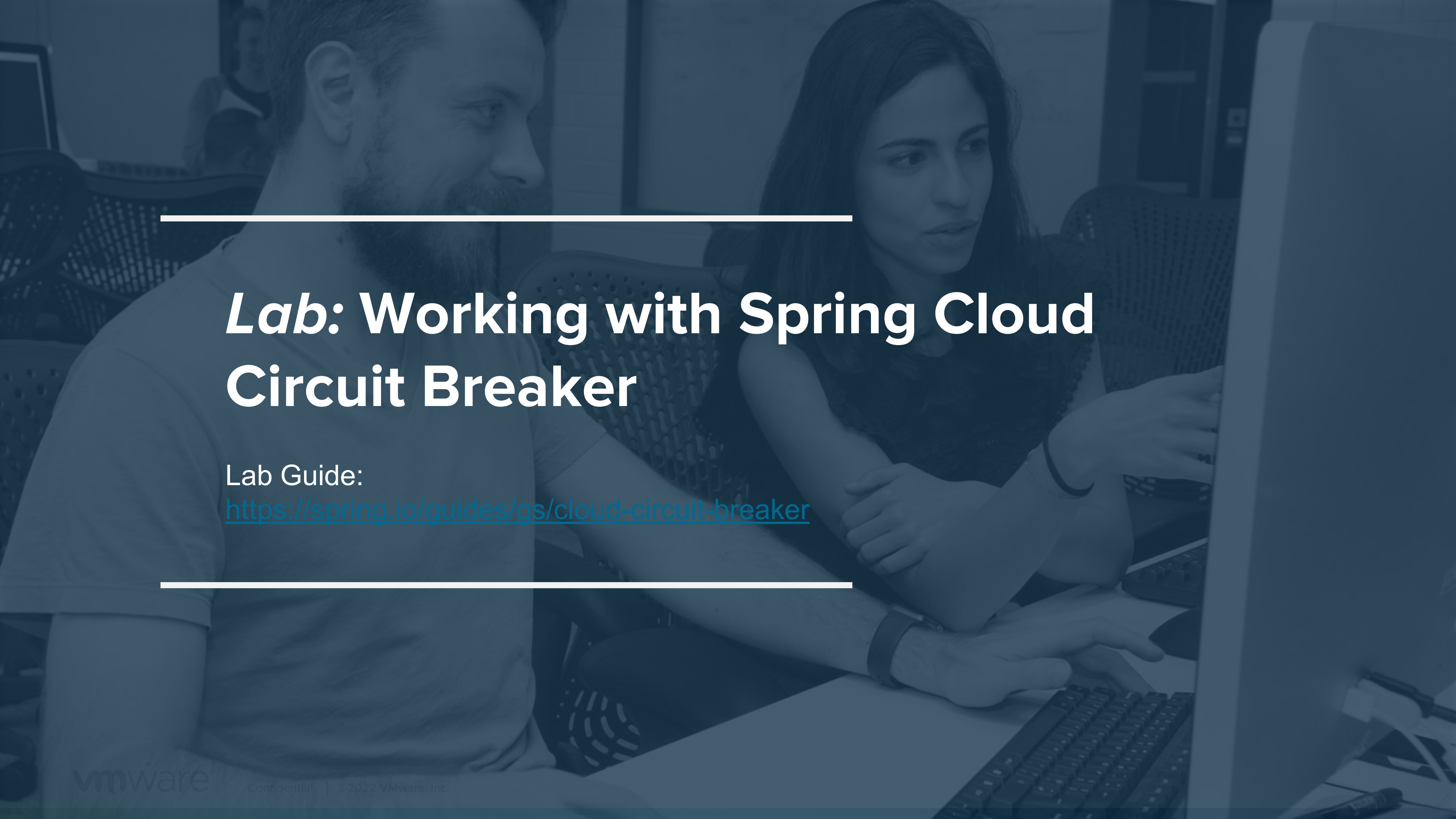
- This can be useful for handling transient failures, such as network timeouts or database connection issues.

Parameter	**Description**	**Default Value**
maxAttempts	Maximum number of retries	3
Interval/wait-duration	Time to wait between retries	5 seconds
waitStrategy	Strategy for calculating wait time	FixedWaitStrategy
ignoreExceptions	List of exceptions to ignore	Empty list

Resilience4J vs Spring Retry

- Both Resilience4J and Spring Retry are powerful tools for implementing retry and circuit breaker patterns in Java applications.
- However, they have distinct approaches and features.

Feature	**Resilience4J**	**Spring Retry**
Resilience Patterns	Circuit Breaker, Retry, Ratelimiter, Bulkhead	Retry
Configuration	Declarative annotations or programmatic configuration	Annotation-based or programmatic configuration
Metrics	Provides built-in metrics	Limited metrics



Lab: Working with Spring Cloud Circuit Breaker

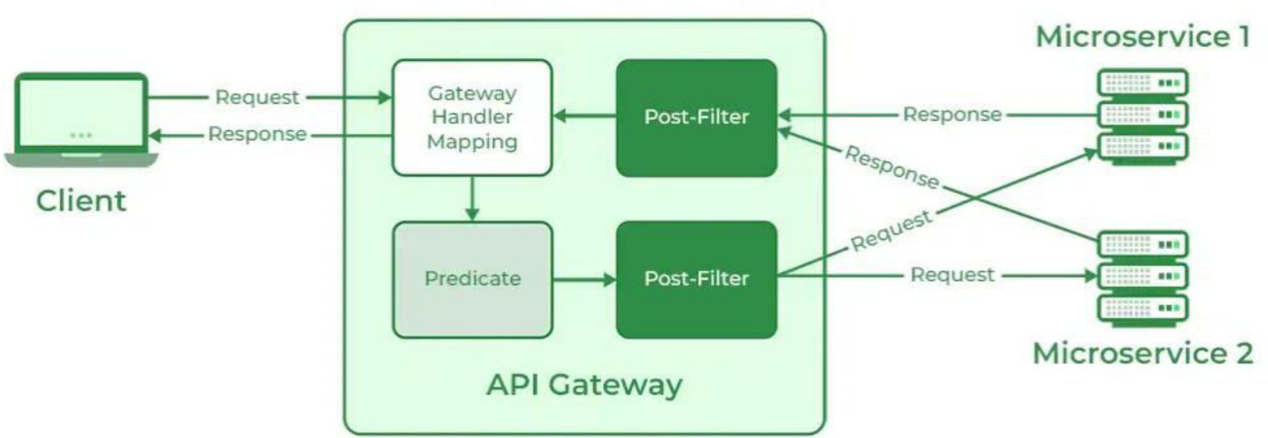
Lab Guide:
https://spring.io/guides/gs/cloud-circuit-breaker

Spring Cloud Gateway

Load balancing

Spring Cloud Gateway

- Spring Cloud Gateway is a powerful API gateway built on Spring Framework 5.0, designed to provide a unified entry point for microservices architectures.
- It acts as a reverse proxy and acts as a single point of entry for all requests to your microservices.



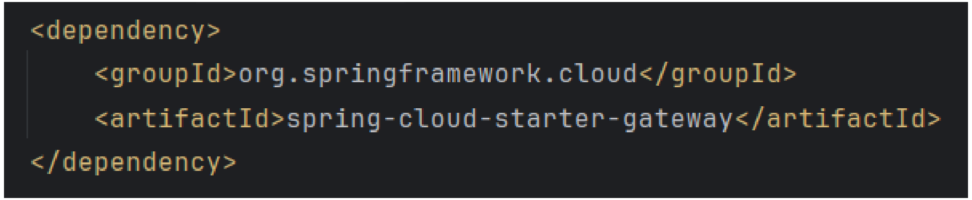
Spring Cloud Gateway

Spring Cloud Gateway features:

- Built on Spring Framework and Spring Boot
- Able to match routes on any request attribute.
- Predicates and filters are specific to routes.
- Circuit Breaker integration.
- Spring Cloud DiscoveryClient integration
- Easy to write Predicates and Filters
- Request Rate Limiting
- Path Rewriting

Spring Cloud Gateway

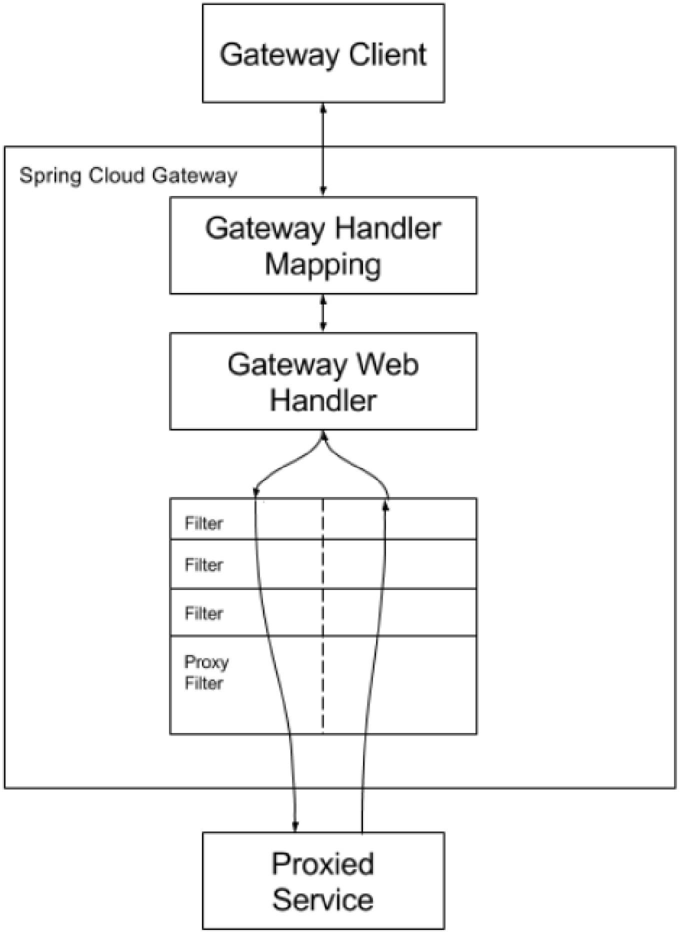
- Dependency needed for Cloud Gateway in Spring Boot:



```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>
```

Spring Cloud Gateway

- Clients make requests to Spring Cloud Gateway. If the Gateway Handler Mapping determines that a request matches a route, it is sent to the Gateway Web Handler.
- This handler runs the request through a filter chain that is specific to the request.
- The reason the filters are divided by the dotted line is that filters can run logic both before and after the proxy request is sent.
- All “pre” filter logic is executed. Then the proxy request is made. After the proxy request is made, the “post” filter logic is run.



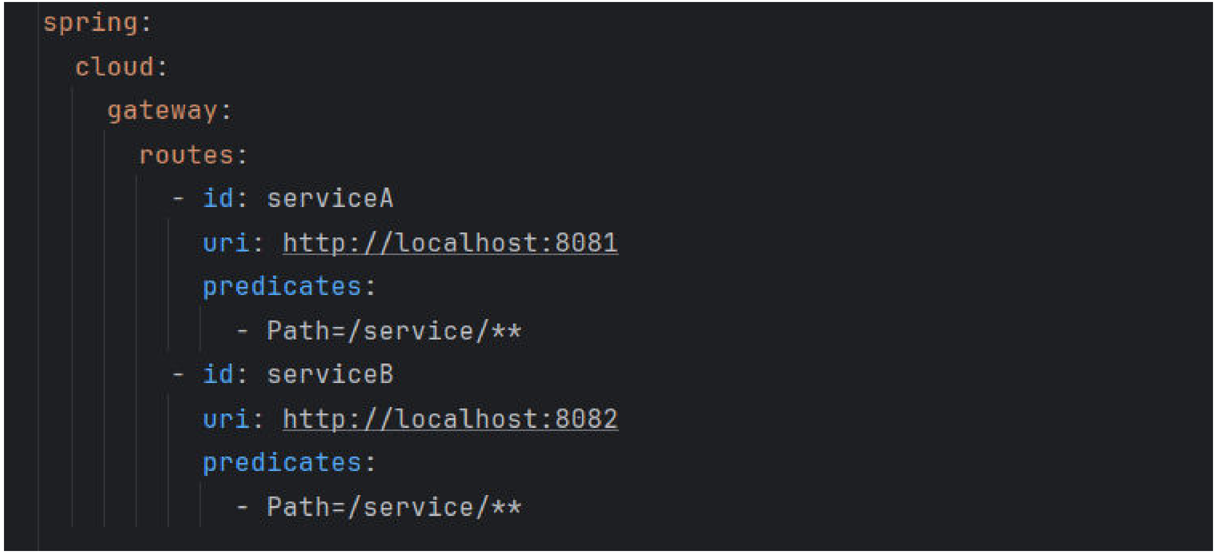
Spring Cloud Gateway

Most important things to consider:

- **Route:** The basic building block of the gateway. It is defined by an ID, a destination URI, a collection of predicates, and a collection of filters. A route is matched if the aggregate predicate is true.
- **Predicate:** This is a Java 8 Function Predicate. The input type is a Spring Framework ServerWebExchange. This lets you match on anything from the HTTP request, such as headers or parameters.
- **Filter:** These are instances of Spring Framework GatewayFilter that have been constructed with a specific factory. Here, you can modify requests and responses before or after sending the downstream request.

Spring Cloud Gateway

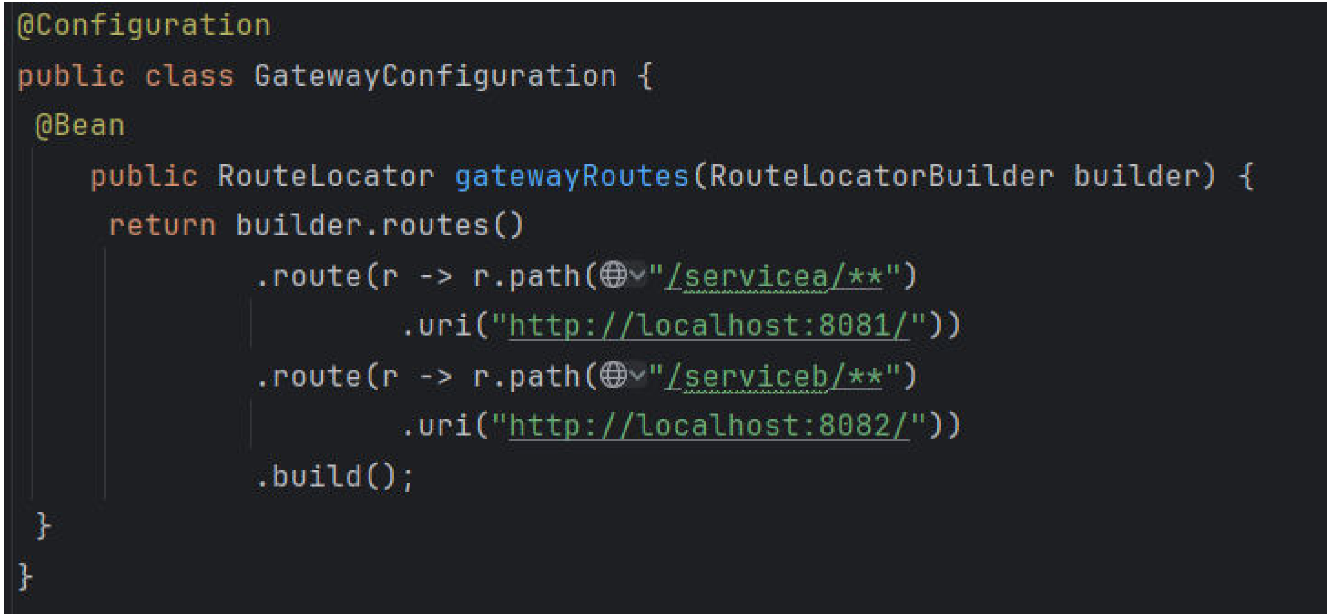
- Gateway Configuration in application.yml:



```yaml
spring:
  cloud:
    gateway:
      routes:
        - id: serviceA
          uri: http://localhost:8081
          predicates:
            - Path=/service/**
        - id: serviceB
          uri: http://localhost:8082
          predicates:
            - Path=/service/**
```

Spring Cloud Gateway

- Gateway Configuration in Java file:



```
@Configuration
public class GatewayConfiguration {
 @Bean
    public RouteLocator gatewayRoutes(RouteLocatorBuilder builder) {
     return builder.routes()
             .route(r -> r.path("/servicea/**")
                     .uri("http://localhost:8081/"))
             .route(r -> r.path("/serviceb/**")
                     .uri("http://localhost:8082/"))
             .build();
 }
}
```

Spring Cloud Gateway

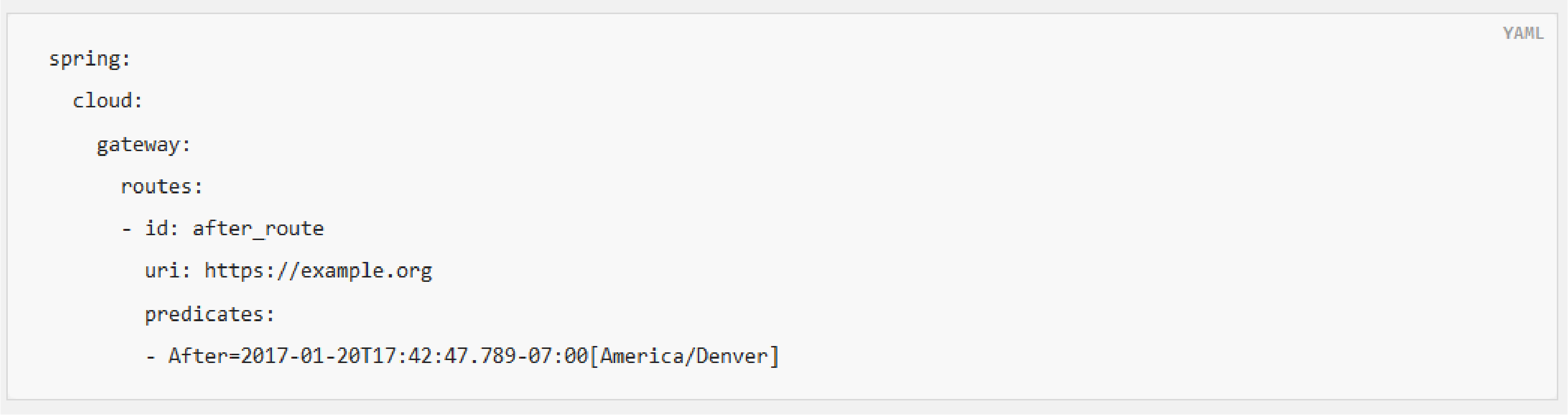
Gateway Route Predicate Factories:

- Spring Cloud Gateway matches routes as part of the Spring WebFlux HandlerMapping infrastructure.
- Spring Cloud Gateway includes many built-in route predicate factories. All of these predicates match on different attributes of the HTTP request.
- You can combine multiple route predicate factories with logical and statements.

Spring Cloud Gateway

After Route Predicate Factory:

- The After route predicate factory takes one parameter, a datetime (which is a java ZonedDateTime).
- This predicate matches requests that happen after the specified datetime.

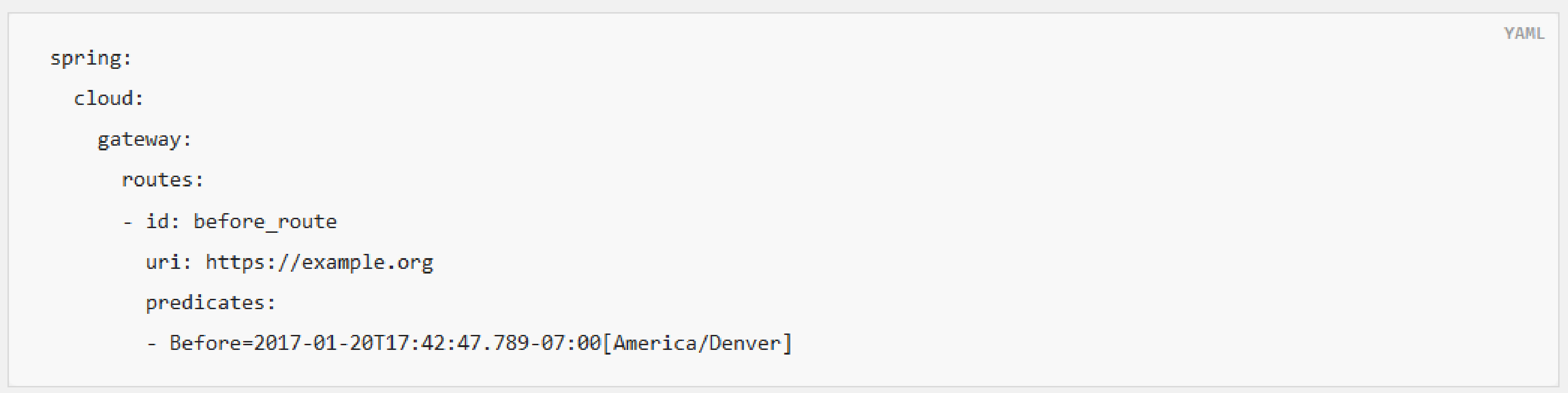


```yaml
spring:
  cloud:
    gateway:
      routes:
      - id: after_route
        uri: https://example.org
        predicates:
        - After=2017-01-20T17:42:47.789-07:00[America/Denver]
```

Spring Cloud Gateway

Before Route Predicate Factory:

- The Before route predicate factory takes one parameter, a datetime (which is a java ZonedDateTime).
- This predicate matches requests that happen before the specified datetime.



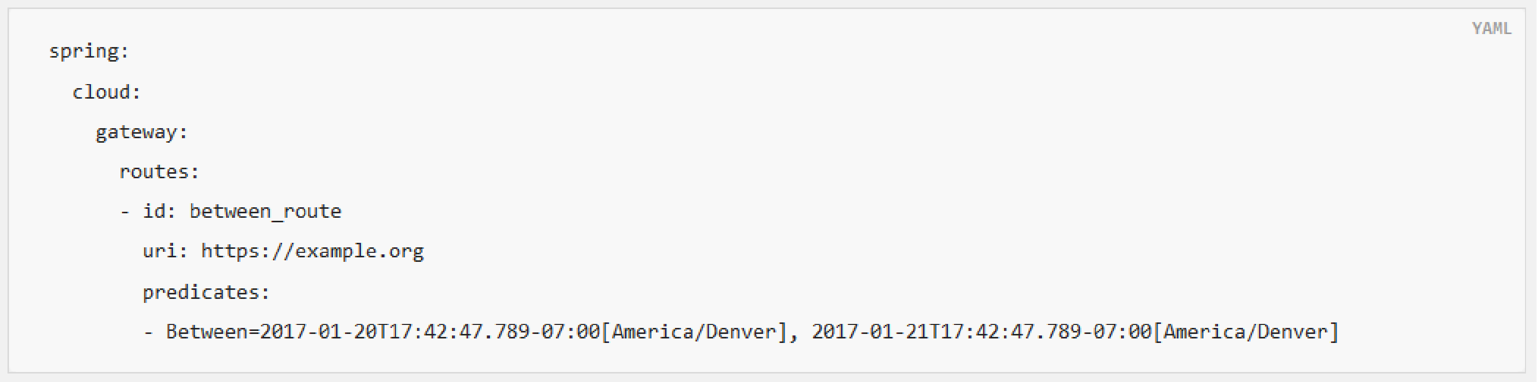
YAML

```yaml
spring:
  cloud:
    gateway:
      routes:
      - id: before_route
        uri: https://example.org
        predicates:
        - Before=2017-01-20T17:42:47.789-07:00[America/Denver]
```

Spring Cloud Gateway

Between Route Predicate Factory:

- The Between route predicate factory takes two parameters, datetime1 and datetime2 which are java ZonedDateTime objects.
- This predicate matches requests that happen after datetime1 and before datetime2. The datetime2 parameter must be after datetime1.



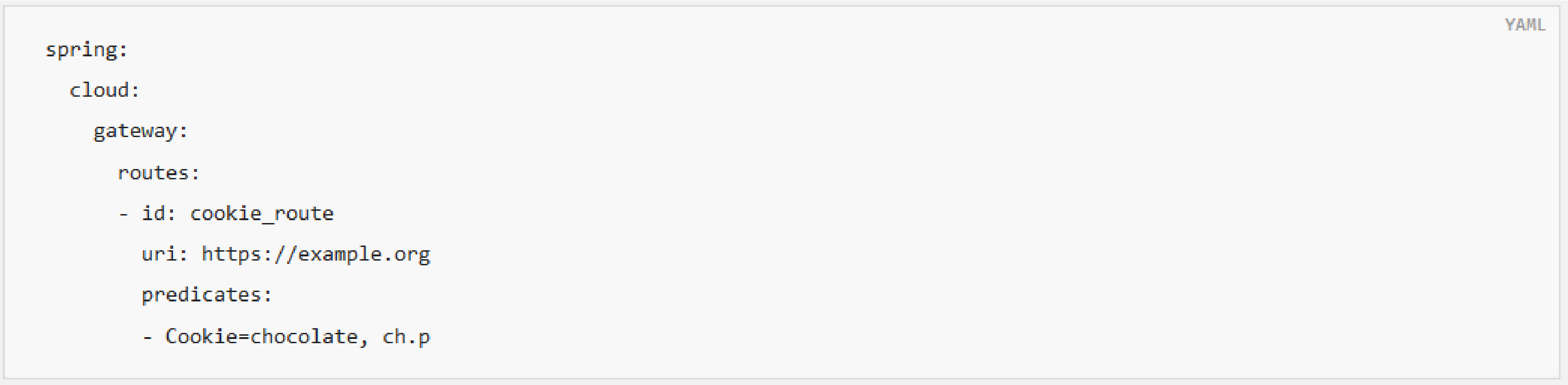
YAML

```yaml
spring:
  cloud:
    gateway:
      routes:
      - id: between_route
        uri: https://example.org
        predicates:
        - Between=2017-01-20T17:42:47.789-07:00[America/Denver], 2017-01-21T17:42:47.789-07:00[America/Denver]
```

Spring Cloud Gateway

Cookie Route Predicate Factory:

- The Cookie route predicate factory takes two parameters, the cookie name and a regexp (which is a Java regular expression).
- This predicate matches cookies that have the given name and whose values match the regular expression.

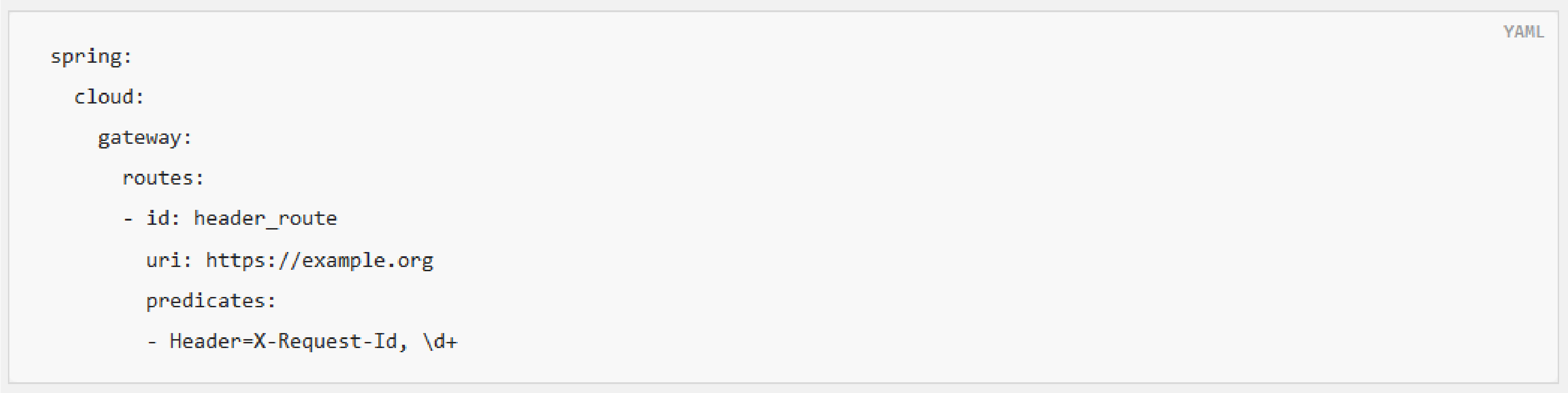


```yaml
spring:
  cloud:
    gateway:
      routes:
      - id: cookie_route
        uri: https://example.org
        predicates:
        - Cookie=chocolate, ch.p
```

Spring Cloud Gateway

Header Route Predicate Factory:

- The Header route predicate factory takes two parameters, the header name and a regexp (which is a Java regular expression).
- This predicate matches with a header that has the given name whose value matches the regular expression.

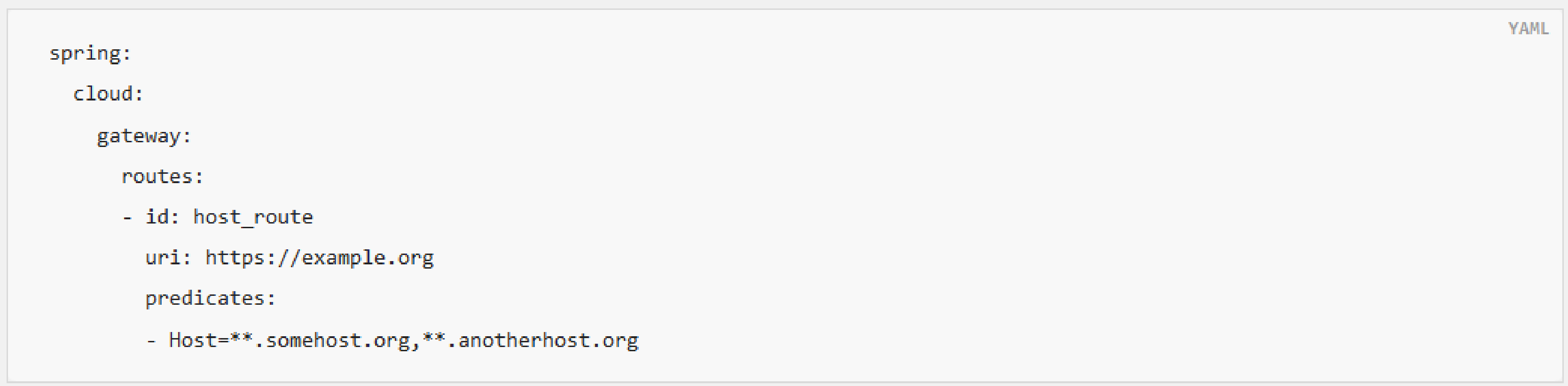


```yaml
spring:
  cloud:
    gateway:
      routes:
      - id: header_route
        uri: https://example.org
        predicates:
        - Header=X-Request-Id, \d+
```

Spring Cloud Gateway

Host Route Predicate Factory:

- The Host route predicate factory takes one parameter: a list of host name patterns. The pattern is an Ant-style pattern with . as the separator.
- This predicates matches the Host header that matches the pattern.

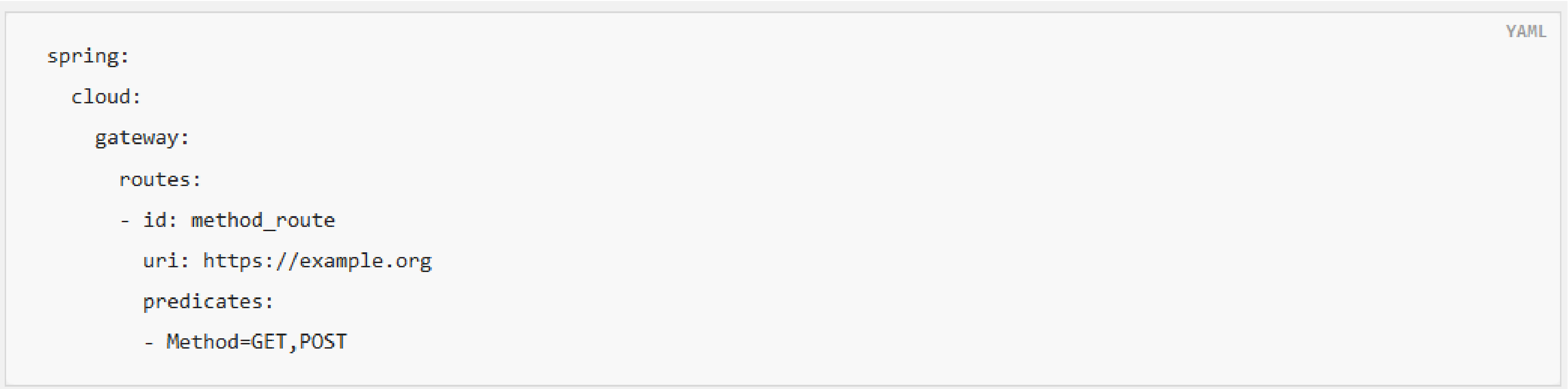


```yaml
spring:
  cloud:
    gateway:
      routes:
      - id: host_route
        uri: https://example.org
        predicates:
        - Host=**.somehost.org,**.anotherhost.org
```

Spring Cloud Gateway

Method Route Predicate Factory:

- The Method Route Predicate Factory takes a methods argument which is one or more parameters: the HTTP methods to match.



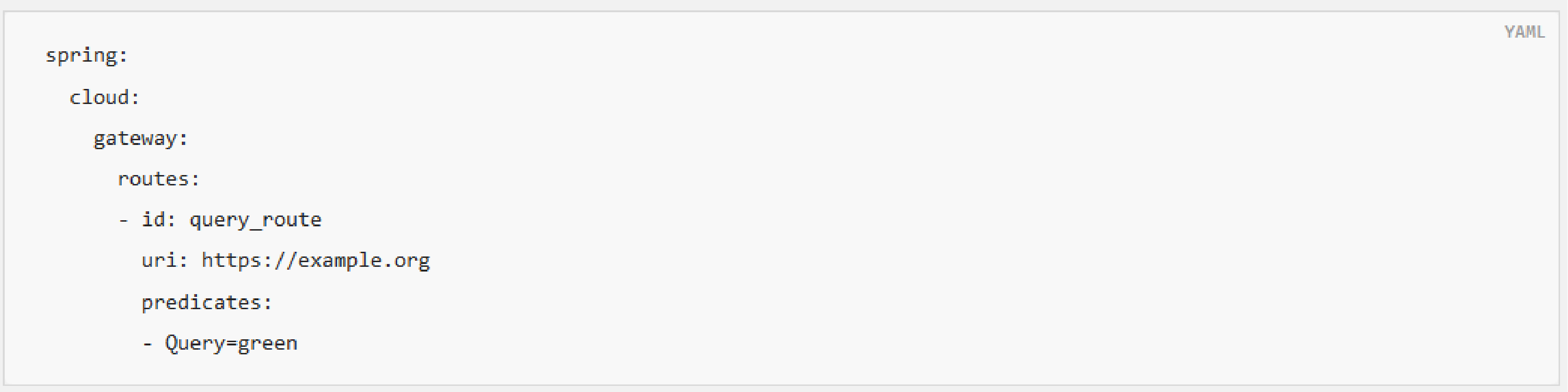
YAML

```yaml
spring:
  cloud:
    gateway:
      routes:
      - id: method_route
        uri: https://example.org
        predicates:
        - Method=GET,POST
```

Spring Cloud Gateway

Query Route Predicate Factory:

- The Query route predicate factory takes two parameters: a required param and an optional regexp (which is a Java regular expression).

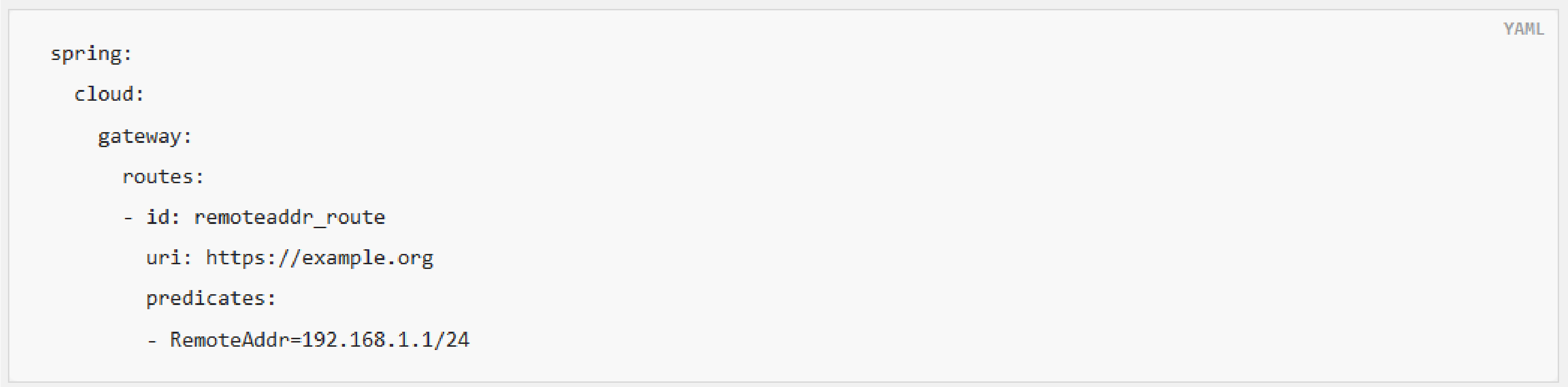


```yaml
spring:
  cloud:
    gateway:
      routes:
      - id: query_route
        uri: https://example.org
        predicates:
        - Query=green
```

Spring Cloud Gateway

RemoteAddr Route Predicate Factory:

- The RemoteAddr route predicate factory takes a list (min size 1) of sources, which are CIDR-notation (IPv4 or IPv6) strings, such as 192.168.0.1/16 (where 192.168.0.1 is an IP address and 16 is a subnet mask).

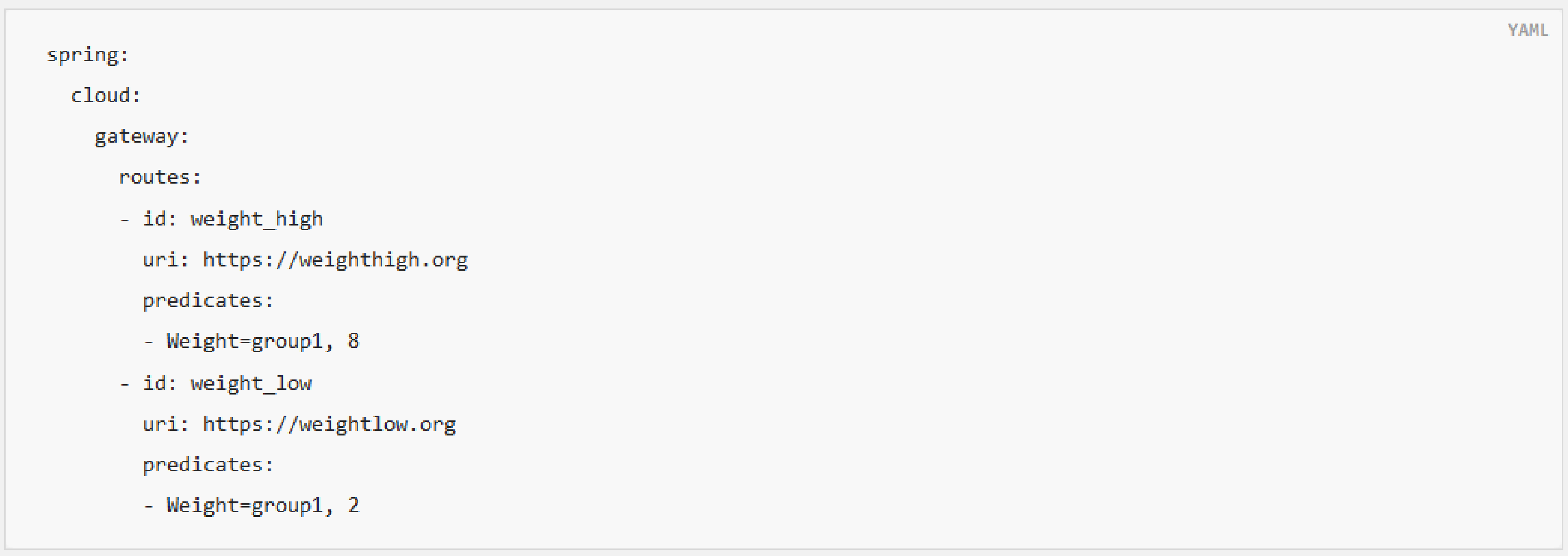


```yaml
spring:
  cloud:
    gateway:
      routes:
      - id: remoteaddr_route
        uri: https://example.org
        predicates:
        - RemoteAddr=192.168.1.1/24
```

Spring Cloud Gateway

Weight Route Predicate Factory:

- The Weight route predicate factory takes two arguments: group and weight (an int). The weights are calculated per group.

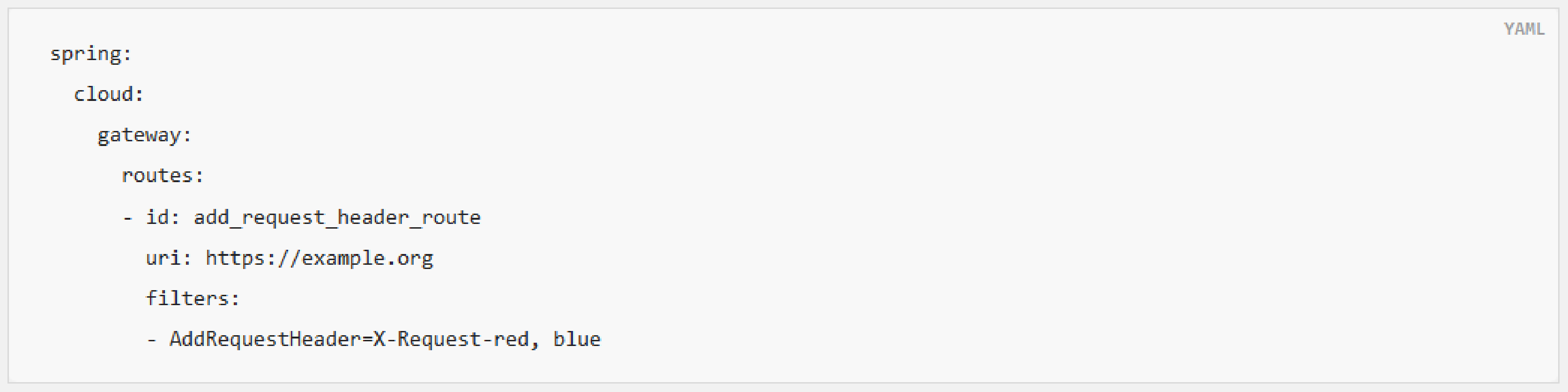


```yaml
spring:
  cloud:
    gateway:
      routes:
      - id: weight_high
        uri: https://weighthigh.org
        predicates:
        - Weight=group1, 8
      - id: weight_low
        uri: https://weightlow.org
        predicates:
        - Weight=group1, 2
```

Spring Cloud Gateway

AddRequestHeader GatewayFilter Factory:

- The AddRequestHeader GatewayFilter factory takes a name and value parameter.
- AddRequestHeader is aware of the URI variables used to match a path or host. URI variables may be used in the value and are expanded at runtime.



YAML

```yaml
spring:
  cloud:
    gateway:
      routes:
      - id: add_request_header_route
        uri: https://example.org
        filters:
        - AddRequestHeader=X-Request-red, blue
```

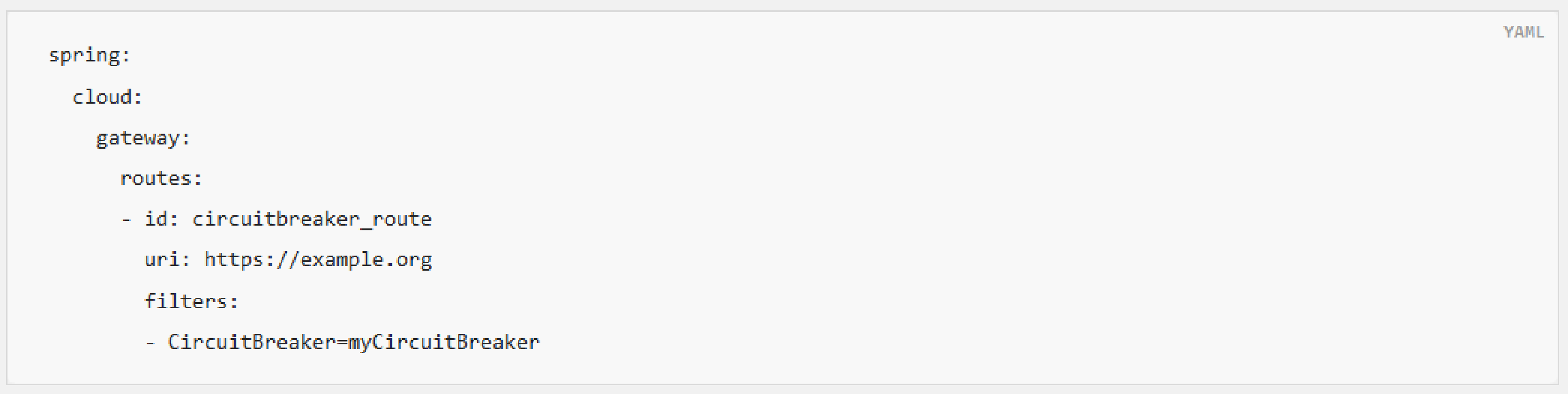
Spring Cloud Gateway

Spring Cloud CircuitBreaker GatewayFilter Factory:

- The Spring Cloud CircuitBreaker GatewayFilter factory uses the Spring Cloud CircuitBreaker APIs to wrap Gateway routes in a circuit breaker.
- Spring Cloud CircuitBreaker supports multiple libraries that can be used with Spring Cloud Gateway. Spring Cloud supports Resilience4J out of the box.
- To enable the Spring Cloud CircuitBreaker filter, you need to place spring-cloud-starter-circuitbreaker-reactor-resilience4j on the classpath.

Spring Cloud Gateway

Spring Cloud CircuitBreaker GatewayFilter Factory:



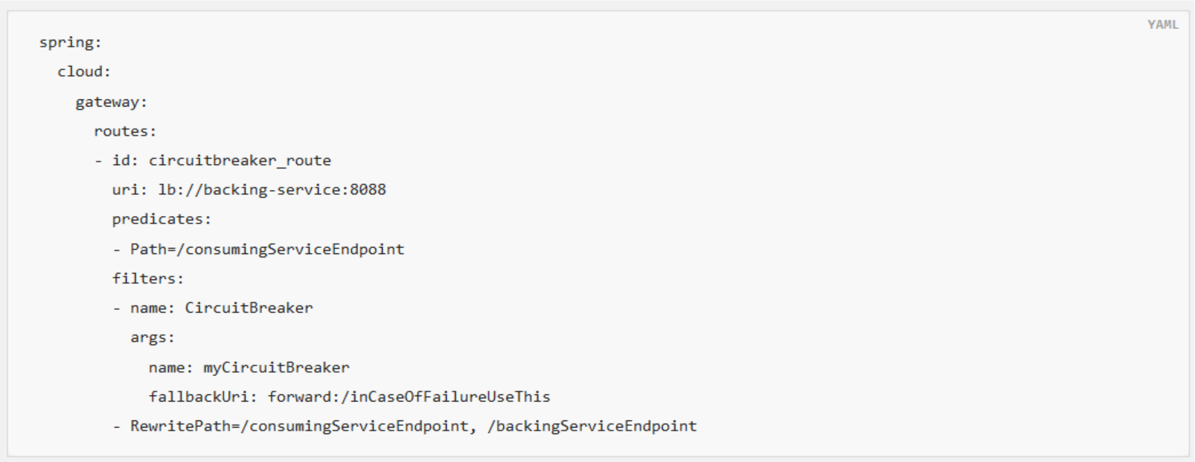
YAML

```
spring:
  cloud:
    gateway:
      routes:
      - id: circuitbreaker_route
        uri: https://example.org
        filters:
        - CircuitBreaker=myCircuitBreaker
```

Spring Cloud Gateway

Spring Cloud CircuitBreaker GatewayFilter Factory:

- The Spring Cloud CircuitBreaker filter can also accept an optional fallbackUri parameter.
- Currently, only forward: schemed URIs are supported. If the fallback is called, the request is forwarded to the controller matched by the URI.

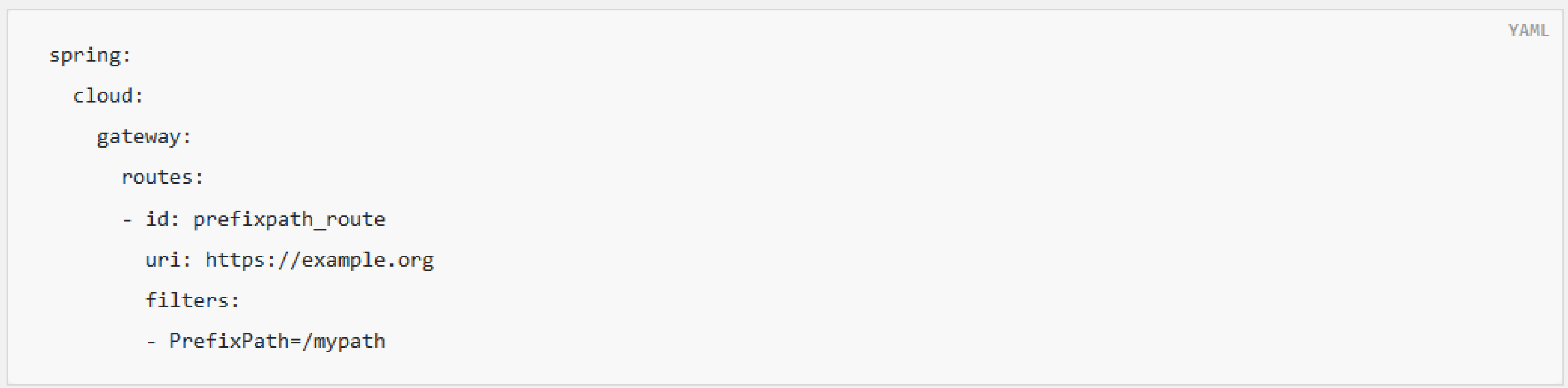


```yaml
spring:
  cloud:
    gateway:
      routes:
      - id: circuitbreaker_route
        uri: lb://backing-service:8088
        predicates:
        - Path=/consumingServiceEndpoint
        filters:
        - name: CircuitBreaker
          args:
            name: myCircuitBreaker
            fallbackUri: forward:/inCaseOfFailureUseThis
        - RewritePath=/consumingServiceEndpoint, /backingServiceEndpoint
```

Spring Cloud Gateway

PrefixPath GatewayFilter Factory:

- The PrefixPath GatewayFilter factory takes a single prefix parameter. The following example configures a PrefixPath GatewayFilter:
- This will prefix /mypath to the path of all matching requests. So a request to /hello would be sent to /mypath/hello.

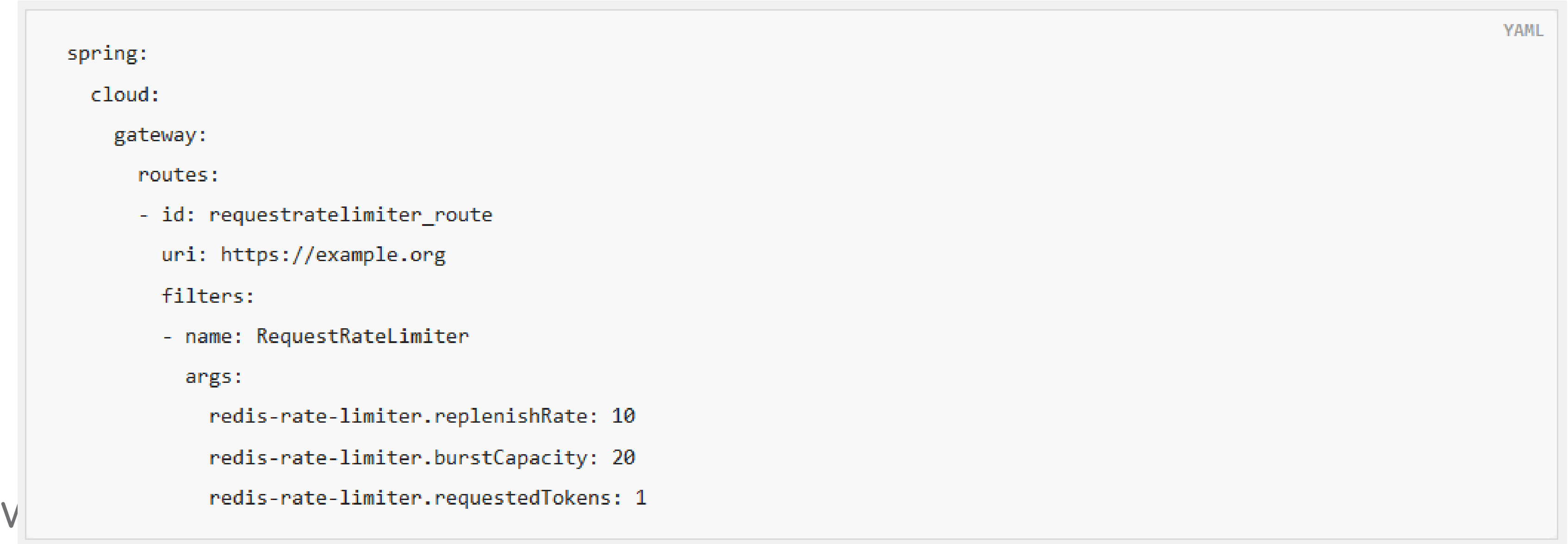


```yaml
spring:
  cloud:
    gateway:
      routes:
      - id: prefixpath_route
        uri: https://example.org
        filters:
        - PrefixPath=/mypath
```

Spring Cloud Gateway

RedisRateLimiter GatewayFilter Factory:

- The Redis implementation is based off of work done at Stripe. It requires the use of the spring-boot-starter-data-redis-reactive Spring Boot starter.



```yaml
spring:
  cloud:
    gateway:
      routes:
      - id: requestratelimiter_route
        uri: https://example.org
        filters:
        - name: RequestRateLimiter
          args:
            redis-rate-limiter.replenishRate: 10
            redis-rate-limiter.burstCapacity: 20
            redis-rate-limiter.requestedTokens: 1
```

Spring Cloud Gateway

RedisRateLimiter GatewayFilter Factory:

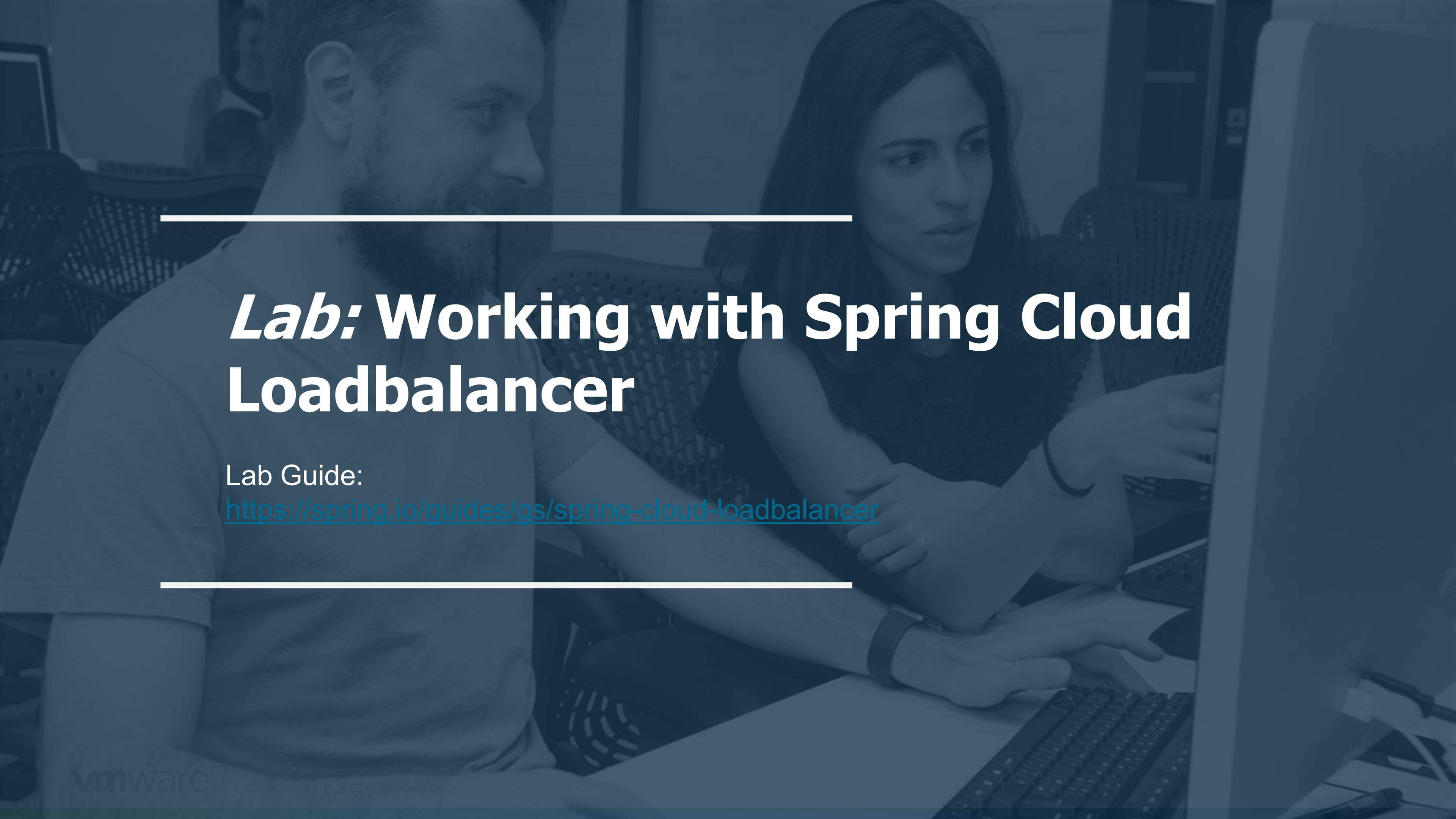
- The **redis-rate-limiter.replenishRate property** is how many requests per second you want a user to be allowed to do, without any dropped requests.
- The **redis-rate-limiter.burstCapacity property** is the maximum number of requests a user is allowed to do in a single second. This is the number of tokens the token bucket can hold. Setting this value to zero blocks all requests.
- The **redis-rate-limiter.requestedTokens property** is how many tokens a request costs. This is the number of tokens taken from the bucket for each request and defaults to 1.

Spring Cloud Gateway

Global GatewayFilter Factory:

The GlobalFilter interface has the same signature as GatewayFilter. These are special filters that are conditionally applied to all routes.

- Forward Routing Filter
- LoadBalancerClient Filter
- ReactiveLoadBalancerClient Filter
- Netty Routing Filter
- Netty Write Response Filter
- RouteToRequestUrl Filter
- WebSocket Routing Filter
- Gateway Metrics Filter



Lab: Working with Spring Cloud Loadbalancer

Lab Guide: https://spring.io/guides/gs/spring-cloud-loadbalancer

Spring Cloud Distributed Tracing using zipkin

Distributed Tracing using Zipkin

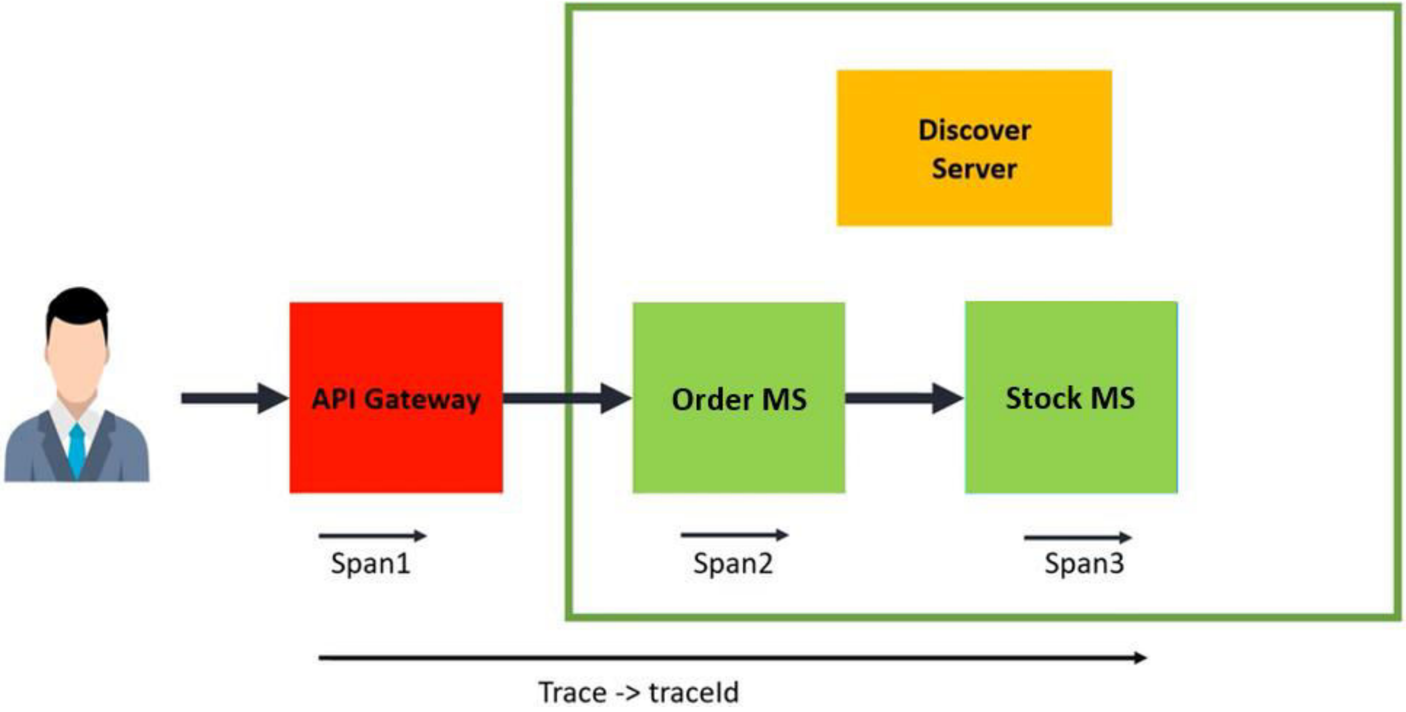
- Distributed tracing is a technique used to profile and monitor applications, especially those built using the microservice architecture.
- Distributed tracing, also called distributed request tracing. IT and DevOps teams can use distributed tracing to monitor applications.
- It identifies the failed microservices or the services having performance issues when there are many services call within a request.
- It is very useful when we need to track the request passing through the multiple microservices. It is also used for measuring the performance of the microservices.

Zipkin

- Zipkin is a distributed tracing system. It helps gather timing data needed to troubleshoot latency problems in service architectures.
- Too many things could happen when a request to an HTTP application is made. A request could include a call to a database engine, to a cache server, or any other dependency like another microservice.
- An application could be sending timing data in the background so that when it’s time to troubleshoot, it can have an integrated view with Zipkin.

Zipkin – Why?

- Consider Flight Ticking Booking Application:



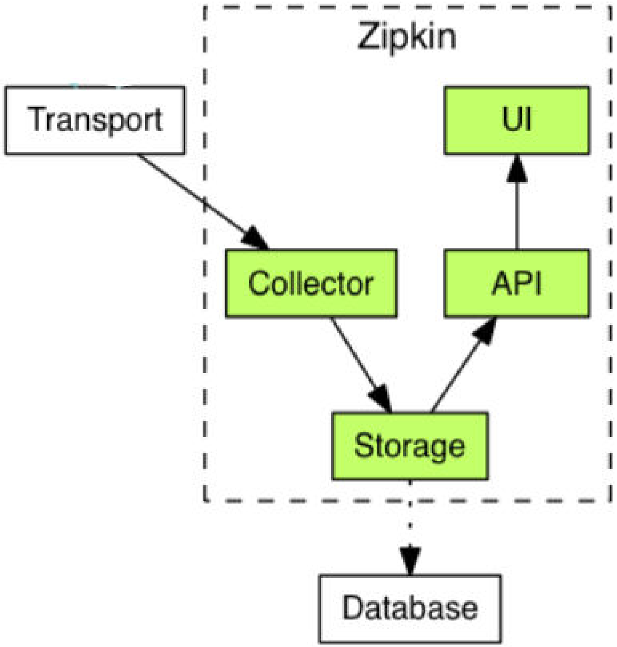
Zipkin – Key Terminologies

- **Trace:** It can be used to represent the whole journey as it propagates through a distributed system and contains multiple spans.
- **Trace ID:** Every trace contains one unique ID that can be named Trace ID, and it can be used to identify the traces uniquely in the Zipkin tracing service.
- **Span:** It can represent a single unit of work within a trace, and it contains information that includes the operation of the request, timing information, metadata, and also the initial request and downstream calls.
- **Span ID:** It can be used to uniquely identify the spans within a trace service, and all spans within the same trace share the same trace ID.
- **Parent Span ID:** It can represent the initial span of the traces, and it can establish the hierarchy of spans in the Zipkin trace service.

Zipkin – Key Terminologies

- **Trace Context:** It can contain the data of the Trace ID, Span ID, and parent Span ID that were propagated along with the request as a traversal through the different services.
- **Sampler:** It is responsible for deciding which requests should be traced and which are not, and it helps to control a subset of requests.
- **Reporter:** It can generate reports of the sending of span data to the tracing backend, and it can store and analyze the traces.
- **Zipkin UI Dashboard:** It can represent the UI dashboard interface of the Zipkin server, and it can visualize traces, analyze performance, and identify issues with the systems.

Zipkin – Architecture



Zipkin - Components

Zipkin Collector

- Once the trace data arrives at the Zipkin collector daemon, it is validated, stored, and indexed for lookups by the Zipkin collector.

Storage

- Zipkin was initially built to store data on Cassandra since Cassandra is scalable, has a flexible schema, and is heavily used within Twitter. However, it made this component pluggable. In addition to Cassandra, it natively support ElasticSearch and MySQL. Other back-ends might be offered as third party extensions.

Zipkin - Components

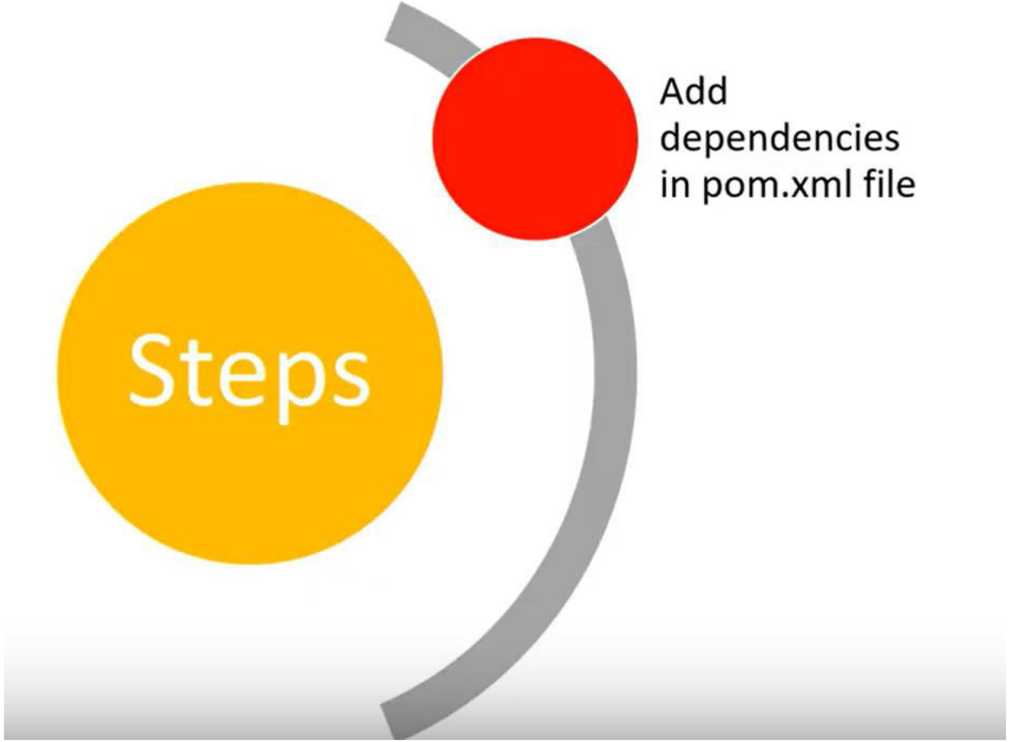
Zipkin Query Service

- Once the data is stored and indexed, we need a way to extract it. The query daemon provides a simple JSON API for finding and retrieving traces. The primary consumer of this API is the Web UI.

Web UI

- We created a GUI that presents a nice interface for viewing traces. The web UI provides a method for viewing traces based on service, time, and annotations. Note: there is no built-in authentication in the UI!

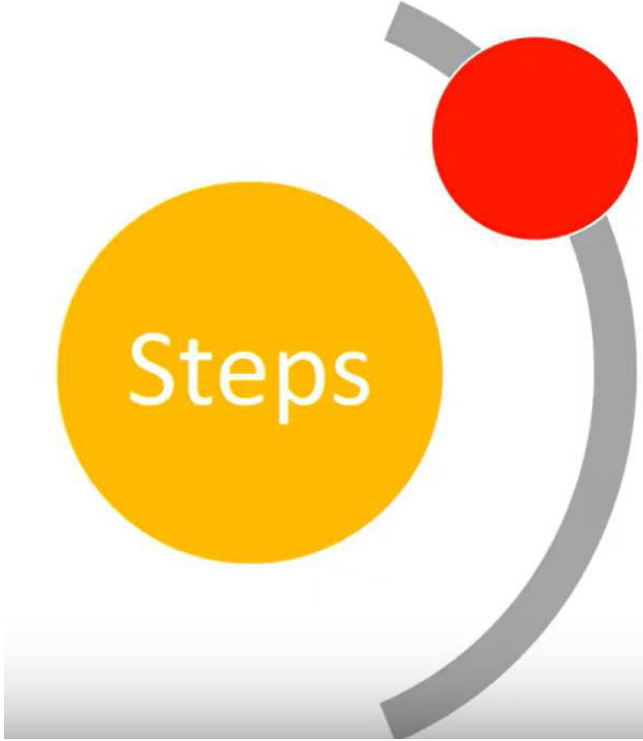
Implement Distributed Tracing using Zipkin



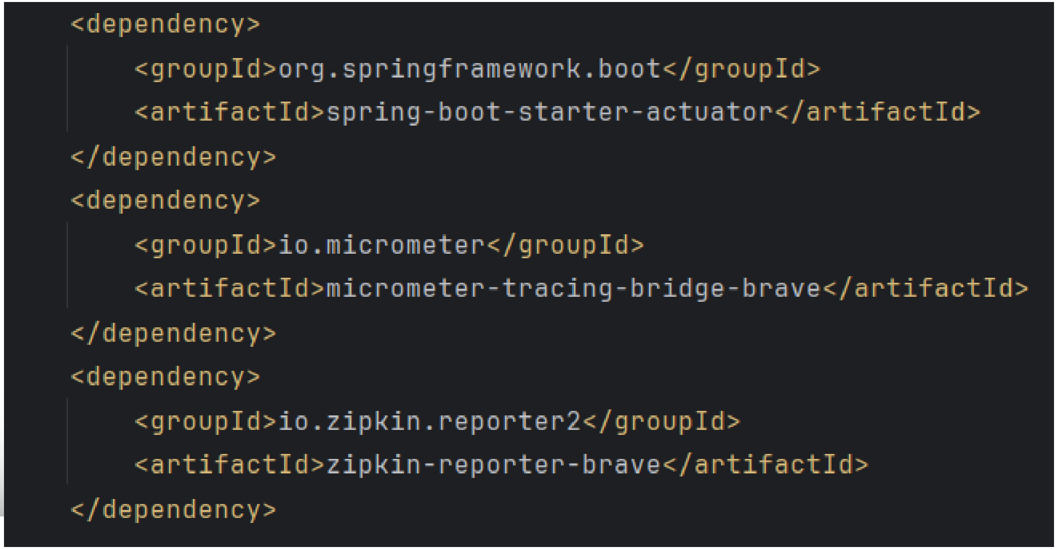
Steps

Add dependencies in pom.xml file

Implement Distributed Tracing using Zipkin

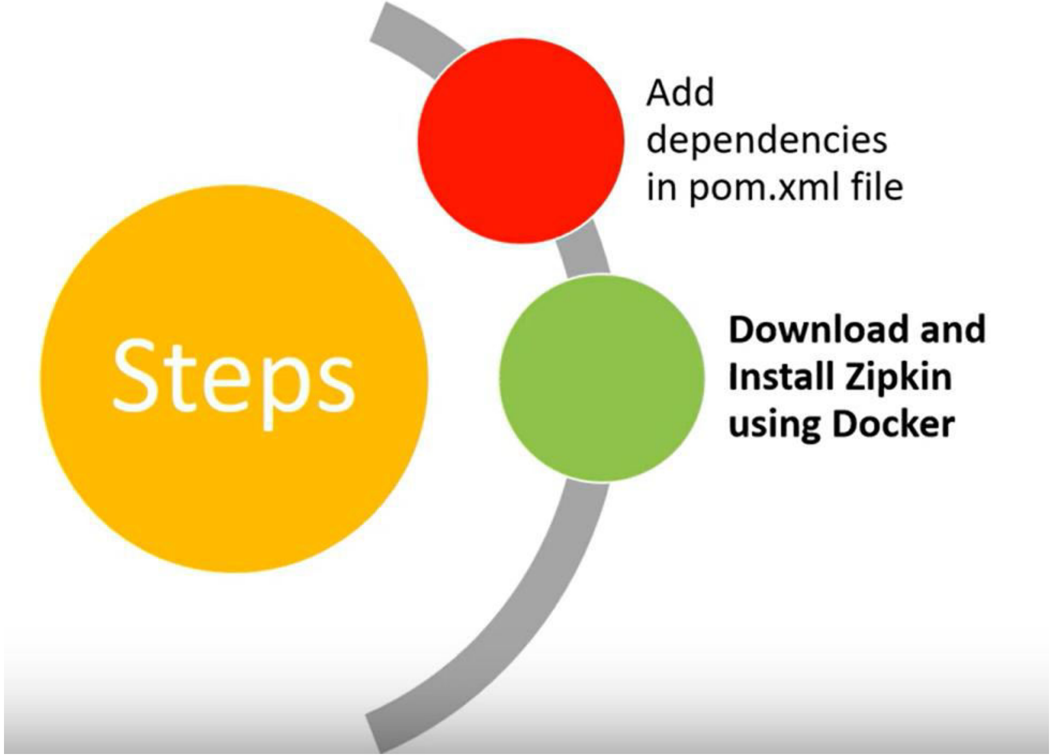


Add
dependencies
in pom.xml file



```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-tracing-bridge-brave</artifactId>
</dependency>
<dependency>
    <groupId>io.zipkin.reporter2</groupId>
    <artifactId>zipkin-reporter-brave</artifactId>
</dependency>
```

Implement Distributed Tracing using Zipkin



Steps

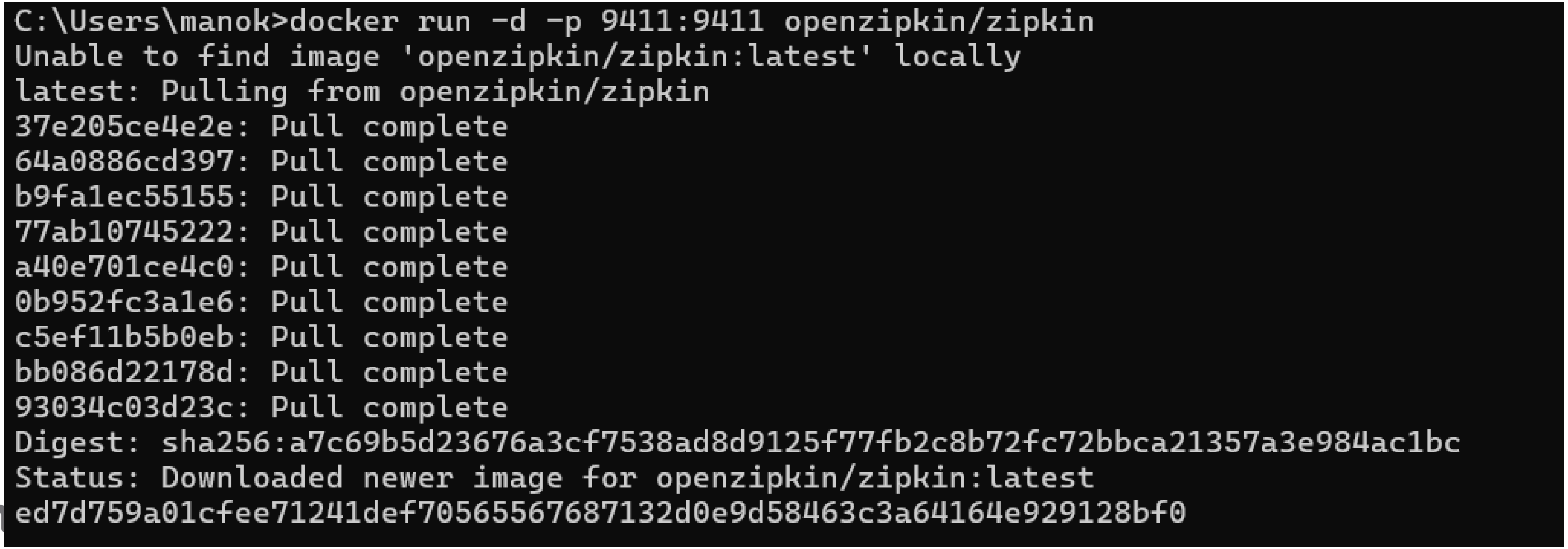
Add dependencies in pom.xml file

Download and Install Zipkin using Docker

Implement Distributed Tracing using Zipkin

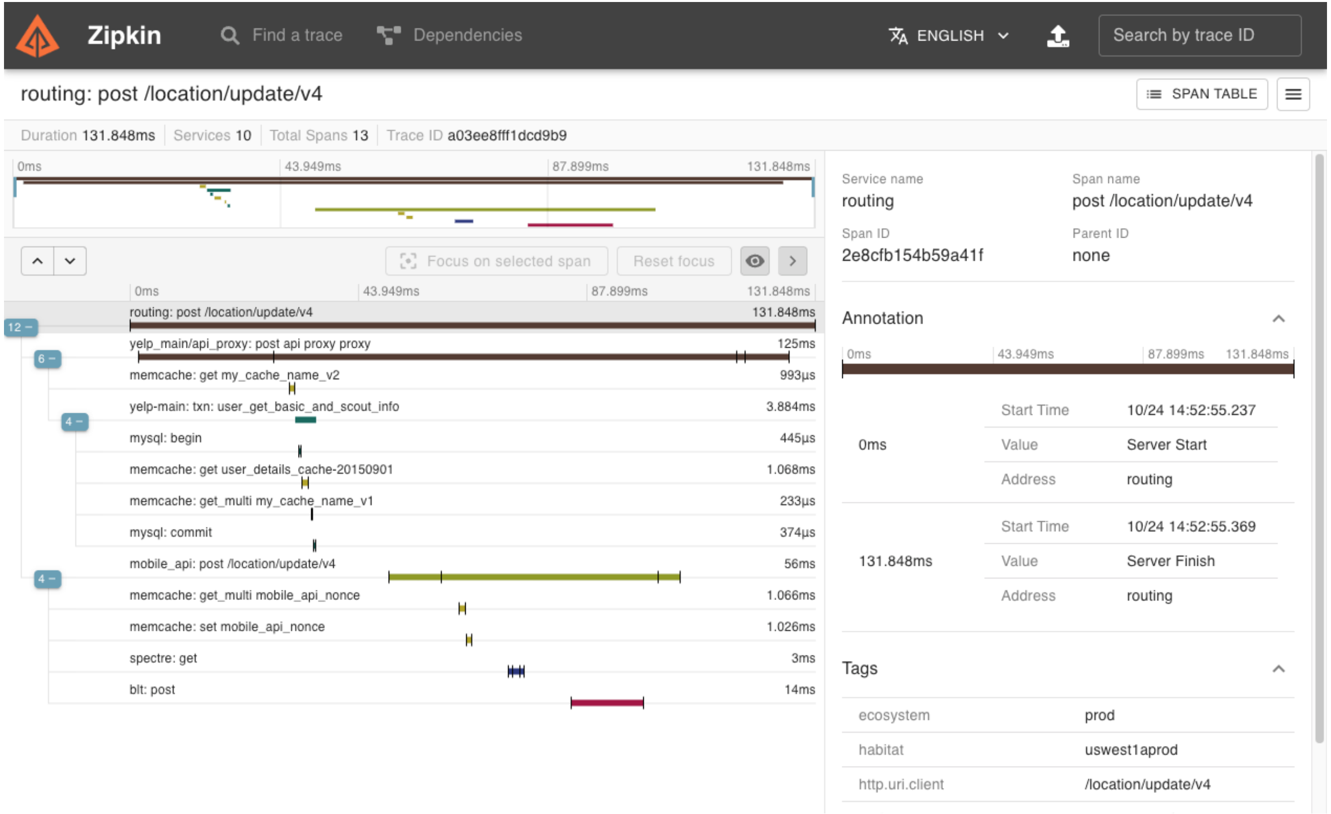
- Zipkin Server application can be installed with your machine using Docker, Java and Homebrew.
- Use Official link to download and install zipkin

  https://zipkin.io/pages/quickstart



```
C:\Users\manok>docker run -d -p 9411:9411 openzipkin/zipkin
Unable to find image 'openzipkin/zipkin:latest' locally
latest: Pulling from openzipkin/zipkin
37e205ce4e2e: Pull complete
64a0886cd397: Pull complete
b9fa1ec55155: Pull complete
77ab10745222: Pull complete
a40e701ce4c0: Pull complete
0b952fc3a1e6: Pull complete
c5ef11b5b0eb: Pull complete
bb086d22178d: Pull complete
93034c03d23c: Pull complete
Digest: sha256:a7c69b5d23676a3cf7538ad8d9125f77fb2c8b72fc72bbca21357a3e984ac1bc
Status: Downloaded newer image for openzipkin/zipkin:latest
ed7d759a01cfee71241def70565567687132d0e9d58463c3a64164e929128bf0
```

Zipkin – UI Console



routing: post /location/update/v4

Duration 131.848ms | Services 10 | Total Spans 13 | Trace ID a03ee8fff1dcd9b9

routing: post /location/update/v4	131.848ms
yelp_main/api_proxy: post api proxy proxy	125ms
memcache: get my_cache_name_v2	993µs
yelp-main: txn: user_get_basic_and_scout_info	3.884ms
mysql: begin	445µs
memcache: get user_details_cache-20150901	1.068ms
memcache: get_multi my_cache_name_v1	233µs
mysql: commit	374µs
mobile_api: post /location/update/v4	56ms
memcache: get_multi mobile_api_nonce	1.066ms
memcache: set mobile_api_nonce	1.026ms
spectre: get	3ms
blt: post	14ms

Service name
routing

Span name
post /location/update/v4

Span ID
2e8cfb154b59a41f

Parent ID
none

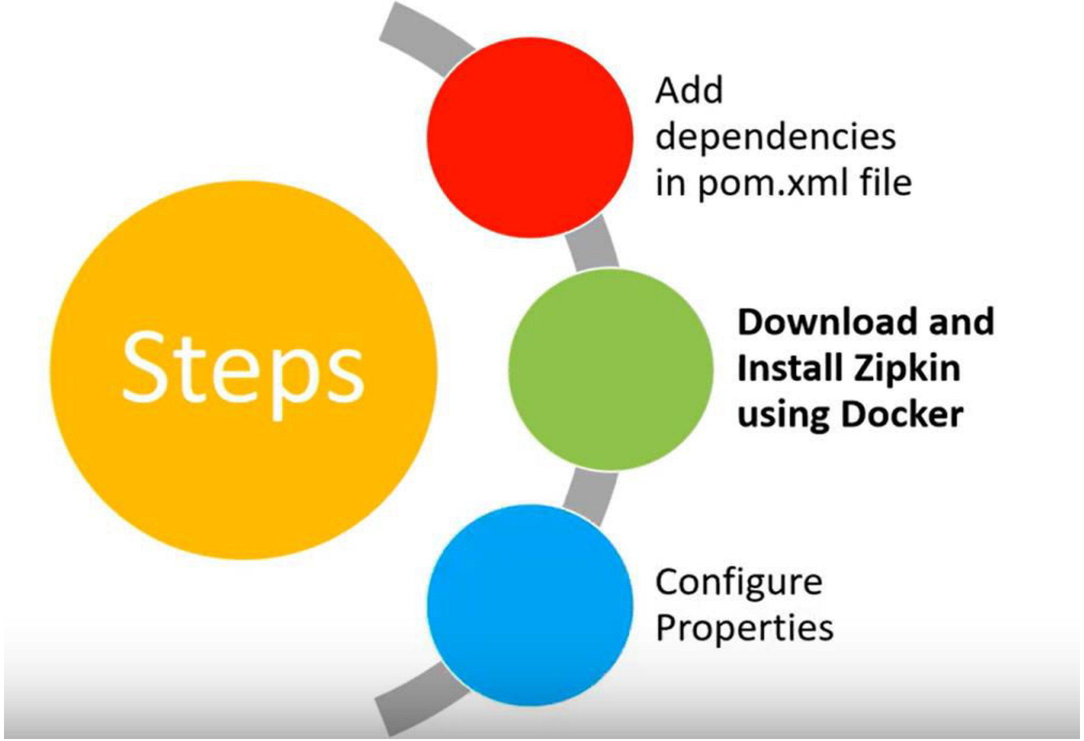
Annotation

0ms	Start Time	10/24 14:52:55.237
	Value	Server Start
	Address	routing
131.848ms	Start Time	10/24 14:52:55.369
	Value	Server Finish
	Address	routing

Tags

ecosystem	prod
habitat	uswest1aprod
http.uri.client	/location/update/v4

Implement Distributed Tracing using Zipkin



Steps

Add dependencies in pom.xml file

Download and Install Zipkin using Docker

Configure Properties

Implement Distributed Tracing using Zipkin

- Zipkin Properties and Configuration

  - management.tracing.sampling.probability = 1.0

If the value is 0 then it will not take any samples for tracing
If the value is 1 then it will take 100% of samples for tracing
If the value is 0.1 then it will take 10% of samples for tracing

Lab: Working with Spring Cloud DT Zipkin

Lab Guide:
https://spring.academy/guides/spring-spring-zipkin

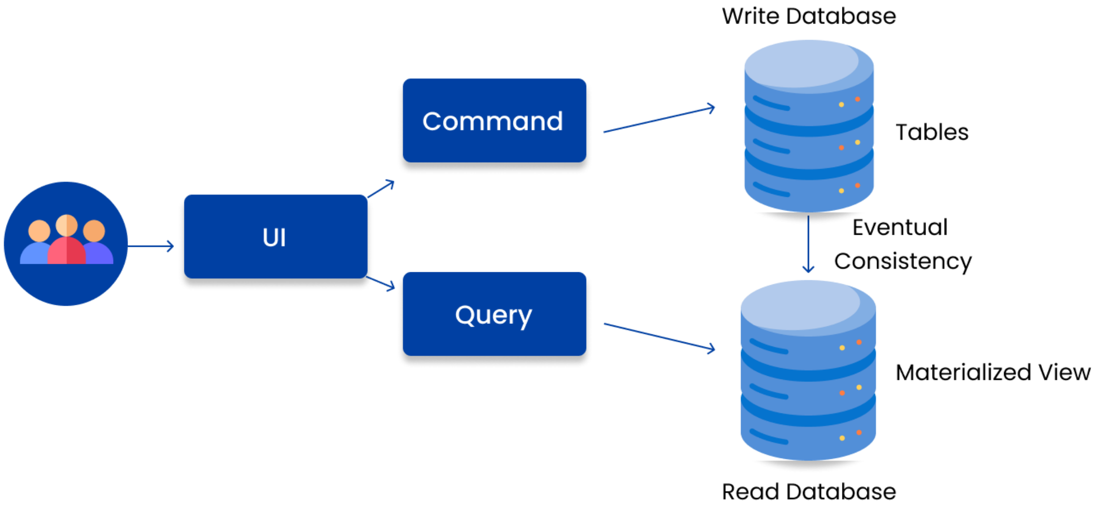
Command Query Responsibility Segregation Pattern

CQRS Pattern

- CQRS is **about segregating the command and query side of the application architecture**.
- CQRS is based on the Command Query Separation (CQS) principle which was suggested by Bertrand Meyer.
- CQS suggests that we divide the operations on domain objects into two distinct categories: Queries and Commands

CQRS Pattern

- The utilization of the event sourcing pattern is common practice in conjunction with it, in order to generate events for every alteration in data.
- Materialized views are maintained in a current state by subscribing to a continuous stream of events.

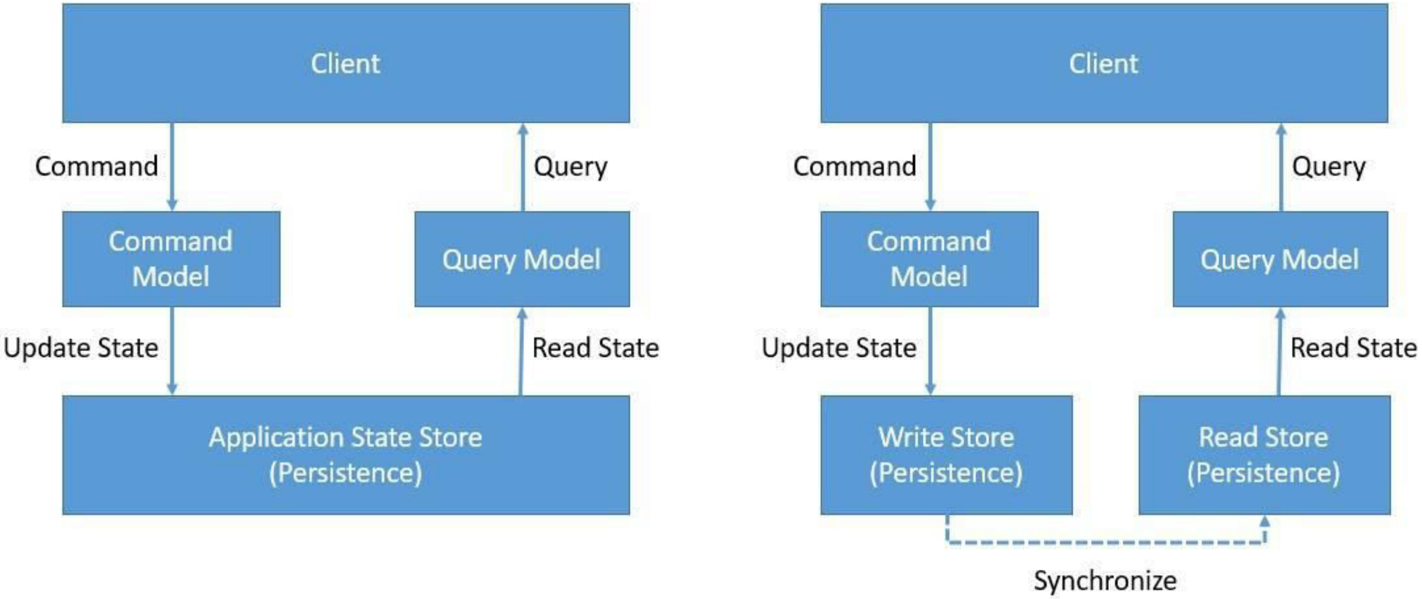


CQRS Pattern

- The implementation of database-per-service necessitates querying that involves retrieving data from multiple services. This requires the combination of data from different services.
- The CQRS pattern suggests splitting the application into two components: the command side and the query side.
  - The command side processes Create, Update, and Delete requests.
  - The query side utilizes materialized views to handle the query component.

CQRS Pattern

- Queries return a result and do not change the observable state of a system. Commands change the state of the system but do not necessarily return a value.



CQRS Pattern

Benefits of CQRS Pattern:

- CQRS provides us **a convenient way to select separate domain models** appropriate for write and read operations; we don’t have to create a complex domain model supporting both

- It helps us to **select repositories that are individually suited** for handling the complexities of the read and write operations, like high throughput for writing and low latency for reading

- It naturally **complements event-based programming models** in a distributed architecture by providing a separation of concerns as well as simpler domain models

CQRS Pattern

Drawbacks of CQRS Pattern:

- Only **a complex domain model can benefit** from the added complexity of this pattern; a simple domain model can be managed without all this

- Naturally **leads to code duplication** to some extent, which is an acceptable evil compared to the gain it leads us to; however, individual judgment is advised

- Separate repositories **lead to problems of consistency**, and it’s difficult to keep the write and read repositories in perfect sync always; we often have to settle for eventual consistency

CQRS Pattern

Synchronization of Command and Query Model:

- **Synchronizing command and query models** in a CQRS architecture is essential to ensure data consistency and prevent discrepancies between the two sides. While CQRS promotes eventual consistency, there are strategies to manage synchronization and minimize delays.

- **Synchronization Strategies:**
  - **Eventual Consistency:** Rely on eventual consistency for updates.
  - **Synchronous Updates:** Update the read side immediately after an event is processed.
  - **Batch Updates:** Update the read side in batches for efficiency.

Event Sourcing

RabbitMQ

vmware

Eventual Consistency using Event Sourcing

- **Event Sourcing** is a pattern that stores changes to the application state as a sequence of events. It's often used in conjunction with CQRS to provide a more flexible and scalable architecture.
- **Key Concepts:**
  - **Events:** Immutable records of changes to the application state.
  - **Event Store:** A specialized data store for storing events.
  - **Event Handlers:** Functions or methods that process events and update the application state.
  - **Eventual Consistency:** The read side data may not be immediately consistent with the write side due to asynchronous updates.

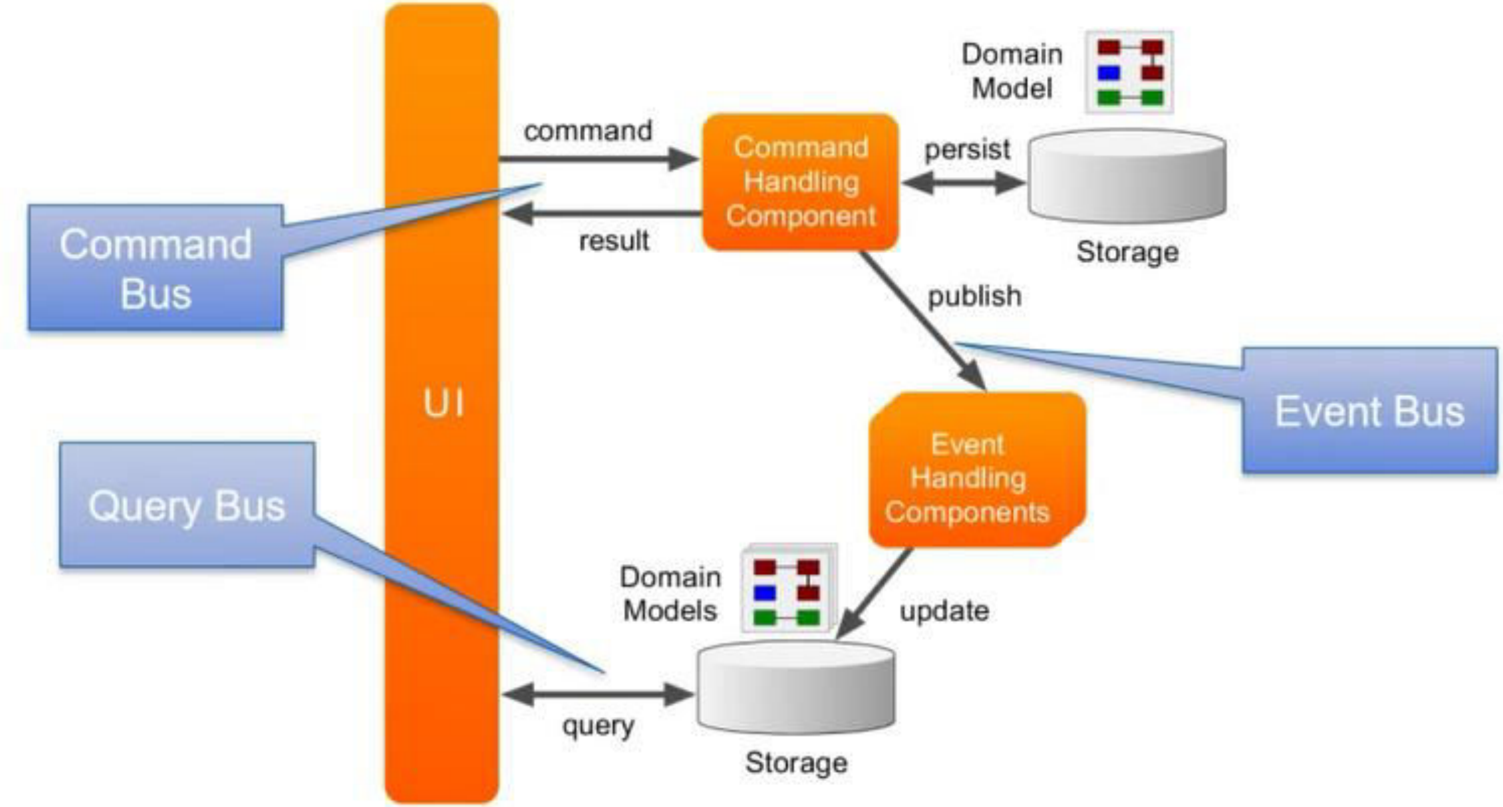
Event Sourcing

- Event sourcing is a powerful distributed system's pattern that records all state changes made to an application in sequence in which the original changes were applied.

- These records serve as system from where current state can be sourced from.

- It also servers as an audit log of all the events that happened within the application over its lifetime.

CQRS Pattern with Event Sourcing



CQRS Pattern with Event Sourcing

- CQRS and Event Sourcing require specific infrastructure require specific infrastructure requirements for storing events (the Event Store) and event transmission (the Messaging Hub) of Commands, Queries, and Events.
- The Axon Framework makes it easier to implement CQRS and event sourcing with minimal boilerplate code and a proven, tested library. The framework provides APIs for easily writing commands, event-sourced aggregates, command handlers, events, event handlers, queries and query handlers.

SAGA Pattern in Microservices

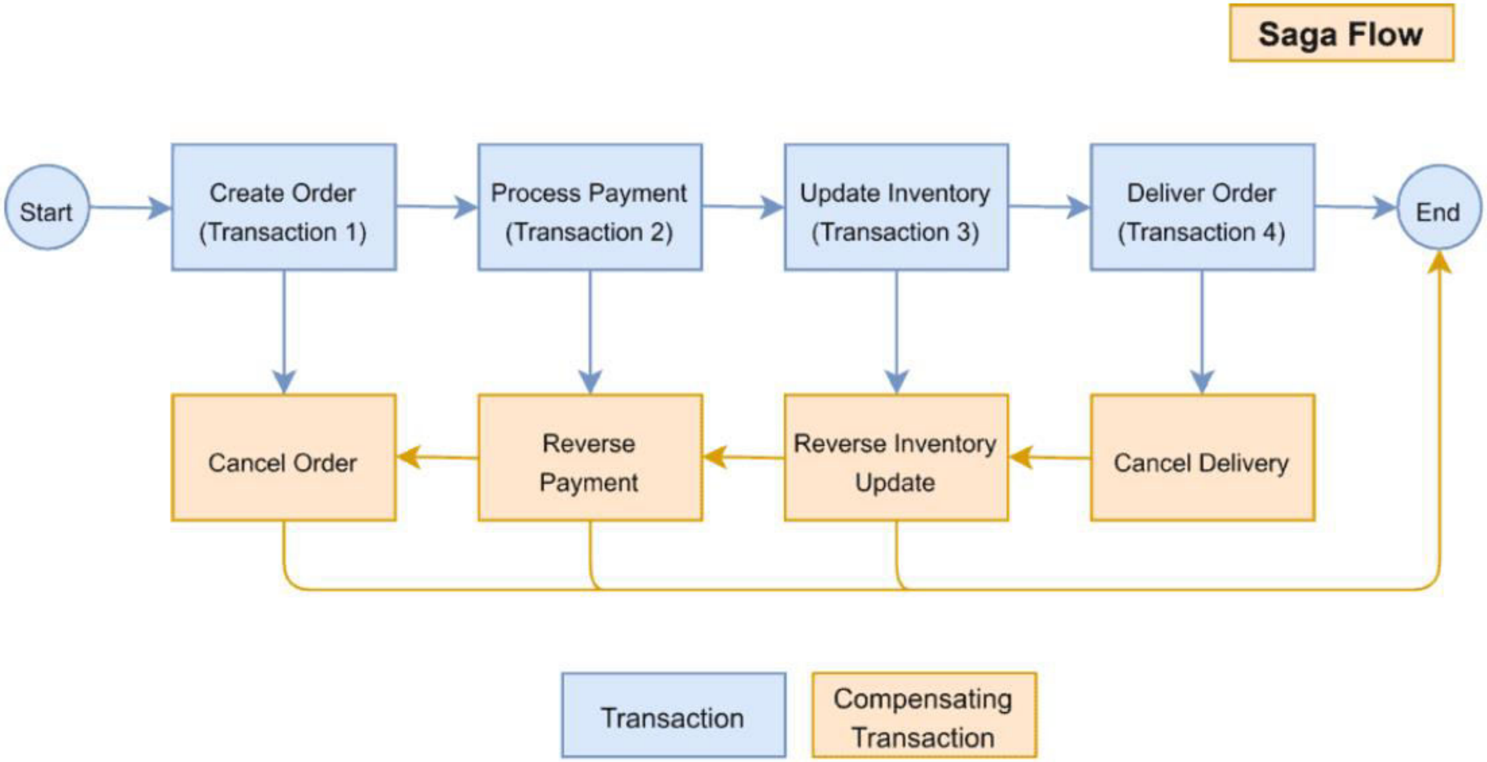
SAGA Pattern

- The saga pattern is based on the idea of breaking a long-running transaction into a series of local transactions, each executed by a different service, and compensating for any failures along the way.
- A saga can be orchestrated by a central service that controls the workflow and triggers the compensation logic, or choreographed by each service that publishes and subscribes to events.
- The saga pattern allows you to achieve eventual consistency and avoid blocking and locking resources, but it also requires careful design and testing of the compensation logic and the event handling.

SAGA Pattern

- The Saga architecture pattern **provides transaction management using a sequence of local transactions.**
- A local transaction is the unit of work performed by a Saga participant. Every operation that is part of the Saga can be rolled back by a compensating transaction.
- Further, the Saga pattern guarantees that either all operations complete successfully or the corresponding compensation transactions are run to undo the work previously completed.

SAGA Pattern


Saga Flow
Start
Create Order (Transaction 1)
Process Payment (Transaction 2)
Update Inventory (Transaction 3)
Deliver Order (Transaction 4)
End
Cancel Order
Reverse Payment
Reverse Inventory Update
Cancel Delivery
Transaction
Compensating Transaction

SAGA Pattern

- To maintain data consistency across services, it is crucial to address the challenge of having separate databases for each service while dealing with business transactions that involve multiple services.
- Compensating requests are triggered by failed requests.
- There are two possible implementation methods:
  - Choreography
  - Orchestration

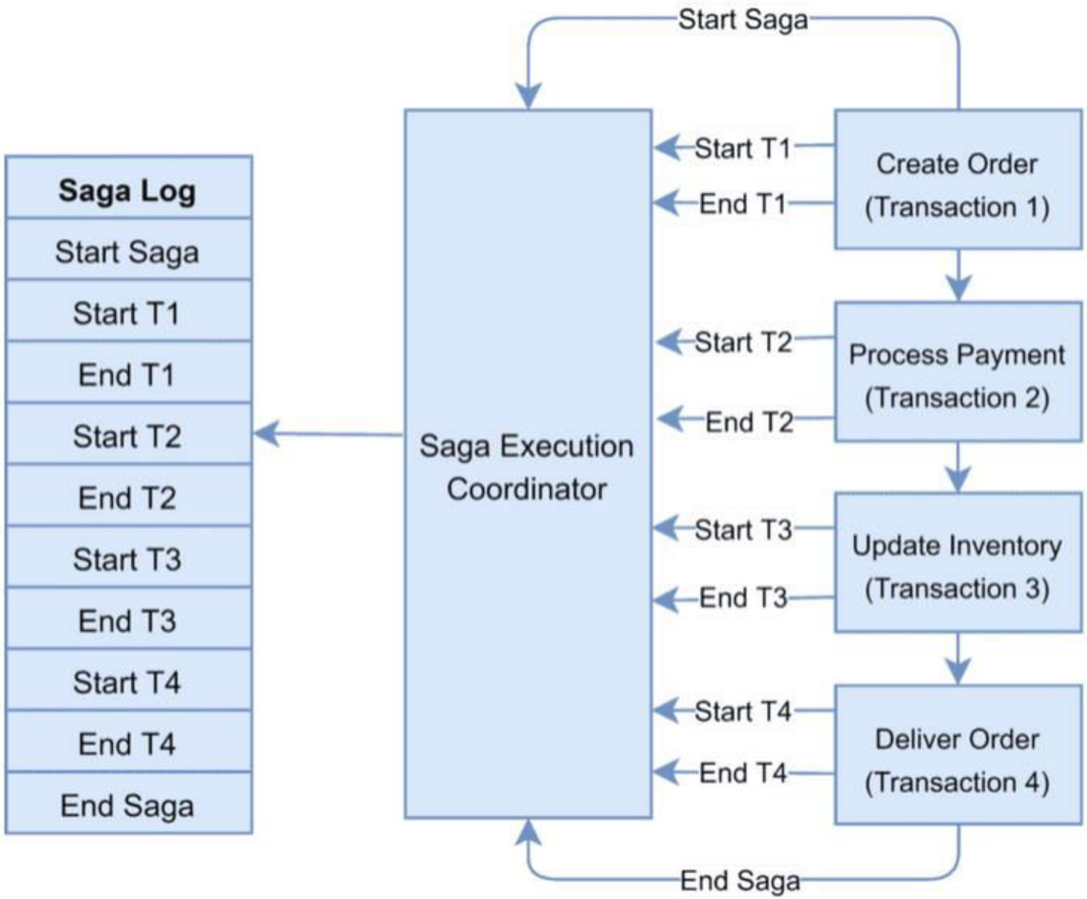
SAGA Choreography Pattern

- The Saga architecture pattern **provides transaction management using a sequence of local transactions.**
- A local transaction is the unit of work performed by a Saga participant. Every operation that is part of the Saga can be rolled back by a compensating transaction. Further, the Saga pattern guarantees that either all operations complete successfully or the corresponding compensation transactions are run to undo the work previously completed.
- In the Saga pattern, **a compensating transaction must be *idempotent* and *retryable*.** These two principles ensure that we can manage transactions without any manual intervention.

SAGA Choreography Pattern

- The **Saga Execution Coordinator is the central component to implement a Saga flow.** It contains a Saga log that captures the sequence of events of a distributed transaction.
- For any failure, the SEC component inspects the Saga log to identify the impacted components and the sequence in which the compensating transactions should run.
- For any failure in the SEC component, it can read the Saga log once it’s coming back up.
- It can then identify the transactions successfully rolled back, which ones are pending, and can take appropriate actions:

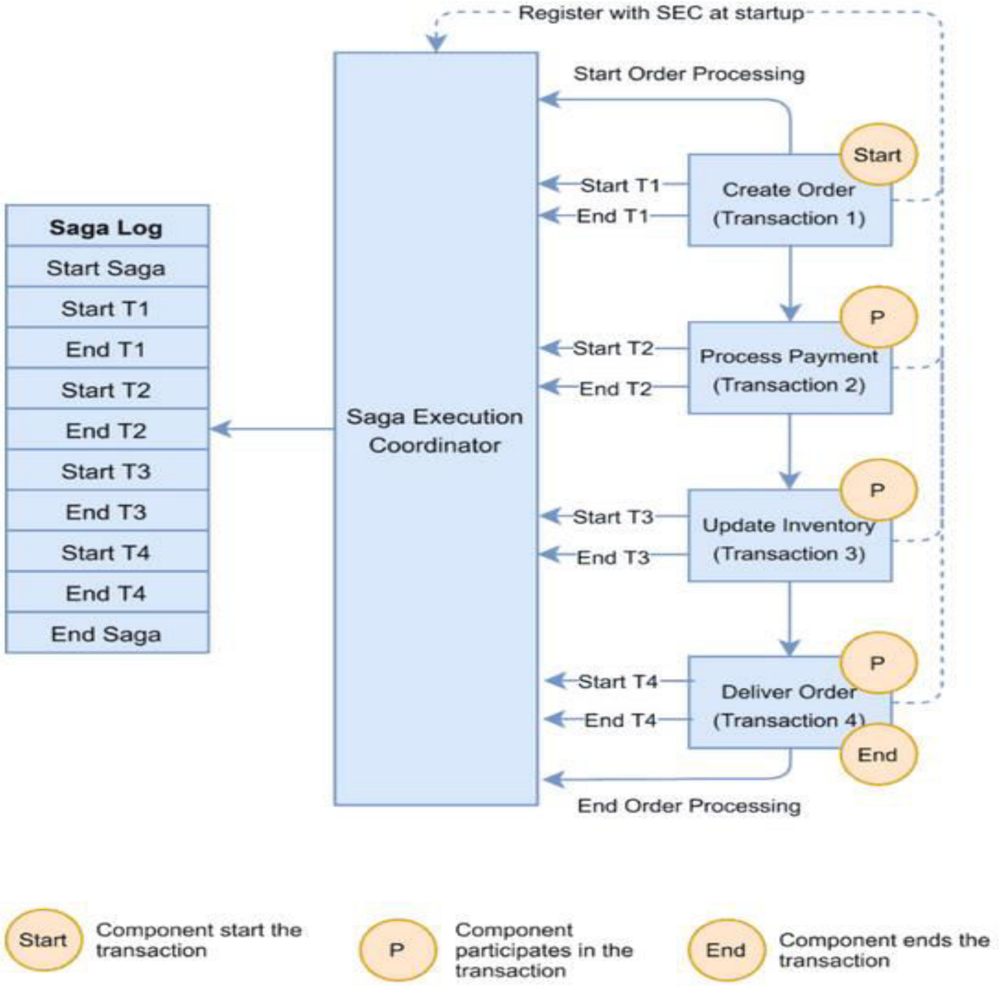
SAGA Pattern



Saga Log
Start Saga
Start T1
End T1
Start T2
End T2
Start T3
End T3
Start T4
End T4
End Saga

Start Saga
Saga Execution Coordinator
Start T1
End T1
Create Order (Transaction 1)
Start T2
End T2
Process Payment (Transaction 2)
Start T3
End T3
Update Inventory (Transaction 3)
Start T4
End T4
Deliver Order (Transaction 4)
End Saga

SAGA Pattern



Register with SEC at startup

Start Order Processing

Saga Log

Start Saga

Start T1

End T1

Start T2

End T2

Start T3

End T3

Start T4

End T4

End Saga

Saga Execution Coordinator

Start T1

End T1

Create Order (Transaction 1)

Start

Start T2

End T2

Process Payment (Transaction 2)

P

Start T3

End T3

Update Inventory (Transaction 3)

P

Start T4

End T4

Deliver Order (Transaction 4)

P

End

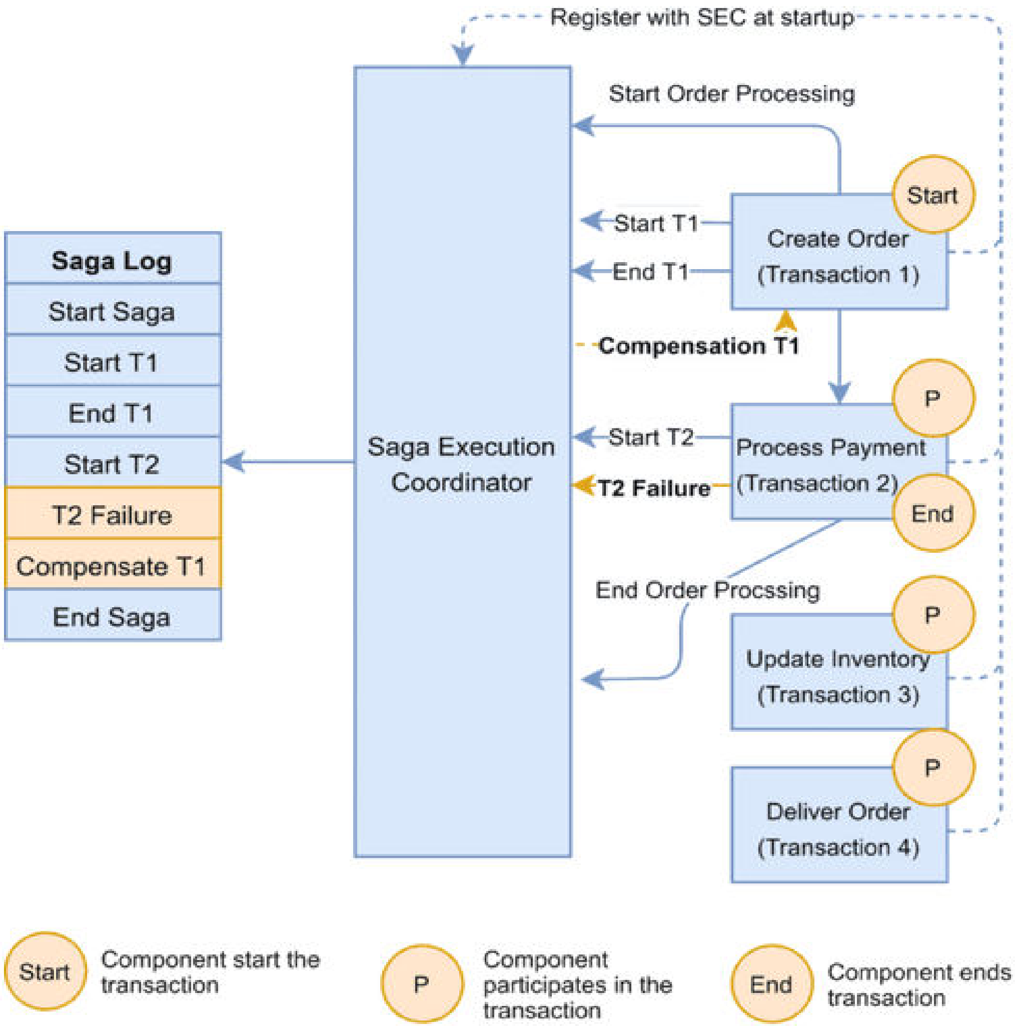
End Order Processing

Start Component start the transaction

P Component participates in the transaction

End Component ends the transaction

SAGA Pattern



Register with SEC at startup
Start Order Processing
Saga Log
Start Saga
Start T1
End T1
Start T2
T2 Failure
Compensate T1
End Saga
Saga Execution Coordinator
Start T1
End T1
Compensation T1
Start T2
T2 Failure
End Order Procssing
Start
Create Order (Transaction 1)
P
Process Payment (Transaction 2)
End
P
Update Inventory (Transaction 3)
P
Deliver Order (Transaction 4)
Start
Component start the transaction
P
Component participates in the transaction
End
Component ends the transaction

SAGA Orchestration Pattern

- In the Saga Choreography pattern, **each microservice that is part of the transaction publishes an event that is processed by the next microservice.**
- To use this pattern, we need to decide if the microservice will be part of the Saga. Accordingly, the microservice needs to use the appropriate framework to implement Saga. In this pattern, the Saga Execution Coordinator is either embedded within the microservice or can be a standalone component.
- In the Saga, choreography flow is successful if all the microservices complete their local transaction, and none of the microservices reported any failure.

SAGA Pattern

Here are a few frameworks available to implement the choreography pattern:

- Axon Saga – a lightweight framework and widely used with Spring Boot-based microservices
- Eclipse MicroProfile LRA – implementation of distributed transactions in Saga for HTTP transport based on REST principles
- Eventuate Tram Saga – Saga orchestration framework for Spring Boot and Micronaut-based microservices
- Seata – open-source distributed transaction framework with high-performance and easy-to-use distributed transaction services

Spring Data Streaming Kafka

Event Streaming

- Event streaming is the practice of capturing data in real-time from event sources like databases, sensors, mobile devices, cloud services, and software applications in the form of streams of events; storing these event streams durably for later retrieval; manipulating, processing, and reacting to the event streams in real-time as well as retrospectively; and routing the event streams to different destination technologies as needed.
- Event streaming thus ensures a continuous flow and interpretation of data so that the right information is at the right place, at the right time.

What I can get from Event Streaming?

- Event streaming is applied to a wide variety of use cases across a plethora of industries and organizations. Its many examples include:
  - To process payments and financial transactions in real-time, such as in stock exchanges, banks, and insurances.
  - To track and monitor cars, trucks, fleets, and shipments in real-time, such as in logistics and the automotive industry.
  - To continuously capture and analyze sensor data from IoT devices or other equipment, such as in factories and wind parks.
  - To collect and immediately react to customer interactions and orders, such as in retail, the hotel and travel industry, and mobile applications.

Apache Kafka

- Apache Kafka is a distributed streaming platform and can be widely used to create real-time data pipelines and streaming applications.
- It can publish and subscribe to records in progress, save these records in an error-free manner, and handle floating records as they arrive.
- Combined with Spring Boot, Kafka can provide a powerful solution for microservice communication and ensure scalability and fault tolerance.

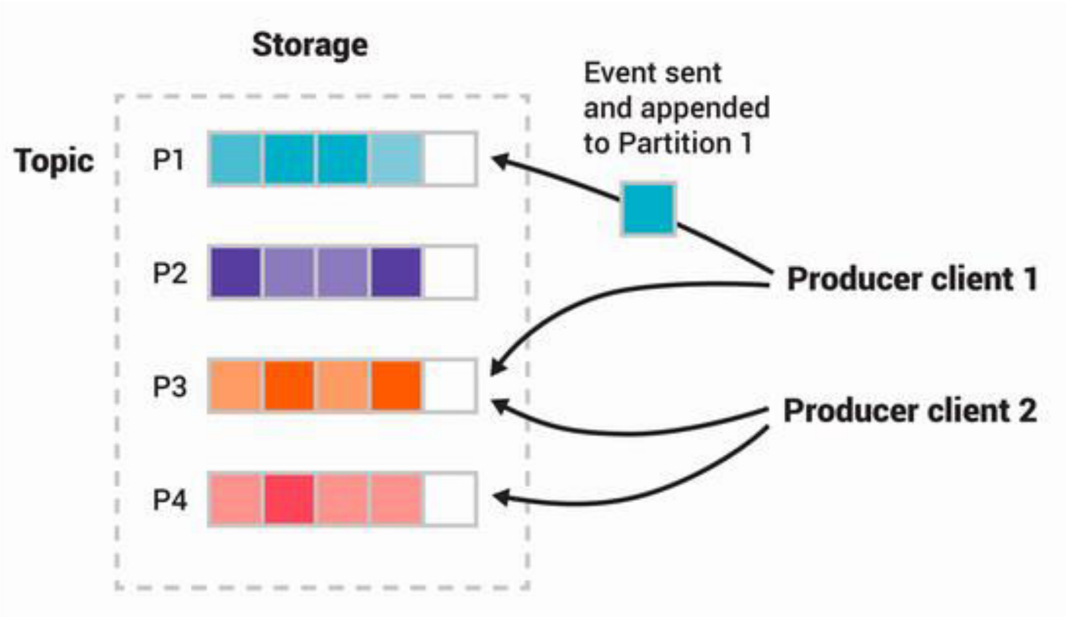
Apache Kafka® is an event streaming platform

- Kafka combines three key capabilities so you can implement your use cases for event streaming end-to-end with a single battle-tested solution:
  - To **publish** (write) and **subscribe to** (read) streams of events, including continuous import/export of your data from other systems.
  - To **store** streams of events durably and reliably for as long as you want.
  - To **process** streams of events as they occur or retrospectively.

Apache Kafka® is an event streaming platform

- **Key Components of Kafka**
  - **Producer**: It sends the messages to the Kafka topic.
  - **Consumer**: It reads the messages from the Kafka topic.
  - **Broker**: Kafka server that stores the messages and serves them to the consumers.
  - **Topic**: The logical channel to which provides the send messages and from which consumers read messages.
  - **Partition**: The topic can be divided into multiple partitions to parallelize processing and storage.
  - **Zookeeper**: It manages and coordinates the Kafka brokers and it can be used in the older versions of Kafka, newer versions use the KRaft mode.

Kafka Message Broker / Kafka Server



Use Cases: Messaging

- Kafka works well as a replacement for a more traditional message broker. Message brokers are used for a variety of reasons (to decouple processing from data producers, to buffer unprocessed messages, etc).
- In comparison to most messaging systems, Kafka has better throughput, built-in partitioning, replication, and fault-tolerance which makes it a good solution for large scale message processing applications.

Use Cases: Web Activity Tracking

- The original use case for Kafka was to be able to rebuild a user activity tracking pipeline as a set of real-time publish-subscribe feeds. This means site activity (page views, searches, or other actions users may take) is published to central topics with one topic per activity type.
- These feeds are available for subscription for a range of use cases including real-time processing, real-time monitoring, and loading into Hadoop or offline data warehousing systems for offline processing and reporting.

Use Cases: Metrics

- Kafka is often used for operational monitoring data.
- This involves aggregating statistics from distributed applications to produce centralized feeds of operational data.

Use Cases: Log Aggregation

- Many people use Kafka as a replacement for a log aggregation solution.
- Log aggregation typically collects physical log files off servers and puts them in a central place (a file server or HDFS perhaps) for processing.
- Kafka abstracts away the details of files and gives a cleaner abstraction of log or event data as a stream of messages.

Use Cases: Streaming and Event Sourcing

- Many users of Kafka process data in processing pipelines consisting of multiple stages, where raw input data is consumed from Kafka topics and then aggregated, enriched, or otherwise transformed into new topics for further consumption or follow-up processing.
- Event sourcing is a style of application design where state changes are logged as a time-ordered sequence of records. Kafka's support for very large stored log data makes it an excellent backend for an application built in this style.