Guia Prático de Microsserviços

Introdução aos Microssserviços

Microsserviços são uma abordagem arquitetural para construir sistemas distribuídos, onde aplicações são divididas em pequenos serviços independentes. Cada serviço é focado em resolver uma funcionalidade específica, sendo gerenciado e implantado de forma isolada.

15 Factor Method

O **15 Factor Method** é uma extensão do conhecido método 12 Factor App, adaptado para atender às necessidades de aplicações modernas baseadas em microsserviços. Abaixo, os fatores são apresentados de forma organizada, com explicações e exemplos práticos.

1. One Codebase, One Application

Princípio:

Uma aplicação deve ter uma única base de código que seja implementada em múltiplos ambientes (desenvolvimento, teste, produção).

Prática:

- Manter o código no mesmo repositório, independente do ambiente.
- Usar ferramentas de controle de versão como Git.

Exemplo:

No GitHub, centralize todos os ambientes em um único repositório. Utilize branches ou tags para diferenciar versões específicas.

2. API First

Princípio:

A API deve ser definida antes do desenvolvimento da aplicação.

Prática:

- Especificar claramente os endpoints, métodos HTTP e respostas.
- Documentar e testar a API para que diferentes equipes possam trabalhar em paralelo.

Exemplo de API para Gerenciamento de Tarefas:

- **Criar uma tarefa**:
    - URL: /tasks

    - Método: POST
    - Entrada: { "title": "string", "description": "string" }
    - Resposta: 201 Created
- **Listar tarefas**:
    - URL: /tasks
    - Método: GET
    - Resposta: 200 OK, retorna lista de tarefas.

3. Dependency Management

Princípio:

Gerencie dependências externas explicitamente, garantindo isolamento e controle total.

Prática com Maven:

- Declare dependências no pom.xml.
- Use repositórios locais para armazenar bibliotecas frequentemente usadas.
- Configure repositórios remotos como o Maven Central para buscar dependências.

4. Design, Build, Release, Run

Princípio:

Divida o ciclo de vida da aplicação em quatro etapas para garantir consistência e previsibilidade.

1. **Design:** Planejar arquitetura, lógica de negócio e APIs.
2. **Build:** Compilar código e criar artefatos como .jar ou .war.
3. **Release:** Configurar o artefato para o ambiente de produção.
4. **Run:** Executar a aplicação, garantindo estabilidade e monitoramento.

5. Configuration, Credentials, Code

Princípio:

Separe configuração, credenciais e código para melhorar a segurança e a portabilidade.

1. **Configuração:** Use variáveis de ambiente para armazenar parâmetros específicos do ambiente.
2. **Credenciais:** Armazene chaves e senhas em sistemas seguros.
3. **Código:** Mantenha o código independente das configurações e credenciais.

Exemplo:

- Configure a URL do banco de dados como variável de ambiente: DB_URL.

- Use ferramentas como HashiCorp Vault para gerenciar segredos.

6. Logs

Princípio:

Centralize os logs para monitorar o comportamento da aplicação e diagnosticar problemas.

Prática:

- Use ferramentas como ELK Stack (Elasticsearch, Logstash e Kibana) ou Grafana.
- Mantenha os logs acessíveis para análise e auditoria.

7. Disposability

Princípio:

A aplicação deve iniciar e desligar rapidamente e de forma confiável.

Prática:

- Garanta que o sistema possa escalar ou se recuperar rapidamente após falhas.
- Use containers para criar instâncias leves e rápidas.

8. Backing Services

Princípio:

Interaja com serviços externos de forma desacoplada e configurável.

Prática:

1. **Tratamento como Recursos Anexados:**
    a. Considere serviços externos (bancos de dados, cache) como plugáveis.
2. **Configuração via Variáveis de Ambiente:**
    a. Configure URLs e credenciais externamente.
3. **Intercambiabilidade:**
    a. Use serviços equivalentes nos diferentes ambientes, como PostgreSQL na produção e SQLite no desenvolvimento.

9. Environment Parity

- **Objetivo**: Reduzir as diferenças entre os ambientes de desenvolvimento, teste e produção para evitar problemas de inconsistência.
- **Práticas Recomendadas**:

    - Use as mesmas versões de bibliotecas, frameworks e infraestrutura em todos os ambientes.
    - Sincronize rapidamente mudanças entre os ambientes.
    - Automatize os processos de build, teste e deploy.
    - Utilize contêineres (como Docker) para garantir consistência entre ambientes.
- **Exemplo**: Se o ambiente de produção usa PostgreSQL, configure o mesmo no ambiente de desenvolvimento.

10. Administrative Processes

- **Definição**: Processos administrativos são tarefas fora do fluxo principal da aplicação, como migrações de banco de dados, monitoramento ou depuração.
- **Práticas Recomendadas**:
    - Execute esses processos em ambientes similares ao de produção.
    - Isole os processos administrativos para não impactar o funcionamento normal da aplicação.
    - Automate comandos administrativos com scripts ou ferramentas dedicadas.
- **Exemplo**:
    - Executar php artisan migrate no Laravel para aplicar mudanças no banco.
    - Utilizar kubectl logs para monitorar serviços em Kubernetes.

11. Port Binding

- **Definição**: A aplicação expõe seus serviços diretamente por uma porta configurável, tornando-se independente de servidores web externos.
- **Práticas Recomendadas**:
    - Inclua um servidor embutido na aplicação (como Tomcat ou Express.js).
    - Configure a porta via arquivos ou variáveis de ambiente, como server.port=8080.
    - Use mapeamento de portas ao executar em contêineres, garantindo flexibilidade.
- **Exemplo**:
    - Uma aplicação Spring Boot expõe suas APIs na porta 8080 por padrão, mas pode ser configurada para usar outra porta com ajustes simples.

12. Stateless Processes

- **Conceito**: Cada requisição à aplicação deve ser independente, sem dependências de estados mantidos no servidor entre interações.
- **Vantagens**:
    - Maior escalabilidade e resiliência.
    - Facilita o balanceamento de carga, pois qualquer instância pode atender uma requisição.
- **Práticas Recomendadas**:

    - Armazene estados compartilhados em bancos de dados ou caches externos, como Redis.
    - Evite armazenar informações da sessão diretamente no servidor.
- **Exemplo**:
    - Um token JWT pode ser usado para autenticação sem depender de informações armazenadas no servidor.

13. Concurrency

- **Definição**: Refere-se à capacidade de processar múltiplas tarefas simultaneamente.
- **Práticas Recomendadas**:
    - Escale horizontalmente adicionando novas instâncias da aplicação para suportar mais requisições.
    - Configure filas de mensagens (RabbitMQ, Kafka) para gerenciar tarefas assíncronas.
- **Exemplo**:
    - Use um cluster Kubernetes para distribuir a carga entre várias instâncias de um microsserviço.

14. Telemetry

- **Definição**: Coleta e monitoramento de dados sobre o desempenho, erros e uso da aplicação em tempo real.
- **Tipos de Monitoramento**:
    - **Logs**: Registros detalhados de eventos específicos (erros, requisições).
    - **Métricas de Telemetria**: Dados contínuos sobre desempenho (latência, uso de CPU).
- **Ferramentas Comuns**:
    - Logs: ELK Stack (Elasticsearch, Logstash, Kibana).
    - Telemetria: Prometheus, Grafana.
- **Exemplo**:
    - Monitore a taxa de erro de APIs ou o tempo médio de resposta para detectar gargalos.

15. Authorization and Authentication

- **Definição**: Gerenciamento de identidade e permissões de usuários.
    - **Autenticação**: Verifica quem é o usuário (ex.: login com e-mail e senha).
    - **Autorização**: Define o que o usuário pode fazer (ex.: permissões de acesso).
- **Práticas Recomendadas**:
    - Use protocolos padrão, como OAuth 2.0, para autenticação segura.
    - Implemente roles (papéis) para gerenciar autorizações.
- **Exemplo**:

- Um usuário autenticado pode acessar a API /users, mas apenas usuários com o papel "Admin" podem acessar /admin.

Organização e Análise Completa da Configuração para Microsserviços com Eureka, Feign, e Spring Cloud

A seguir, a descrição completa e organizada dos arquivos application.yml para cada serviço dentro de uma arquitetura de microsserviços, com a utilização do **Eureka Server**, **Eureka Client**, **Spring Cloud Load Balancer**, e **OpenFeign**.

Arquitetura de Microsserviços

Padrões a serem seguidos:

1. **Descoberta de Serviços**: Os microsserviços podem localizar outros microsserviços automaticamente, utilizando um serviço central de registro, sem a necessidade de URLs fixas.
2. **Registro de Serviços**: Cada microsserviço se registra automaticamente com um nome único no servidor central (Eureka).
3. **Balanceamento de Carga**: O tráfego é distribuído entre várias instâncias de um microsserviço, melhorando o desempenho e a escalabilidade.

Funções do Eureka:

- O **Eureka Server** atua como um servidor de descoberta de serviços.
- Microsserviços (como accounts e loans) se registram no **Eureka Server** e se comunicam entre si usando apenas o nome do serviço (ex: http://accounts).
- O **balanceamento de carga** é gerenciado automaticamente pelo Eureka.

Configuração de Microsserviço accounts

Arquivo accounts.yml:

```
server:
  port: 8080

spring:
  application:
    name: "accounts"  # Nome do serviço
  profiles:
    active: "prod"  # Perfil ativo, neste caso, produção
  datasource:
    url: jdbc:mysql://localhost:3306/accountsdb  # URL do banco de dados
    username: root  # Usuário do banco de dados
    password: root  # Senha do banco de dados
  jpa:
```

```
    show-sql: true  # Exibe as queries SQL no log
  sql:
    init:
      mode: always  # Inicializa o banco de dados toda vez que a aplicação sobe
  config:
    # Conexão com o servidor de configuração
    import: "optional:configserver:http://localhost:8083/"
management:
  endpoints:
    web:
      exposure:
        include: "*"  # Exposição de todos os endpoints do Actuator
  endpoint:
    shutdown:
      enabled: true  # Habilita o endpoint de shutdown
  info:
    env:
      enabled: true  # Exibe informações sobre o ambiente

eureka:
  instance:
    preferIpAddress: true  # Usa o IP em vez do hostname (útil no ambiente local)
  client:
    fetchRegistry: true  # O cliente Eureka busca o registro de serviços
    registerWithEureka: true  # O serviço se registra no Eureka
    serviceUrl:
      defaultZone: http://localhost:8070/eureka/  # URL do Eureka Server

info:
  app:
    name: "accounts"  # Nome do aplicativo
    description: "Eazy Bank Accounts Application"  # Descrição do aplicativo
    version: "1.0.0"  # Versão do aplicativo
```

Configuração do Config Server

Arquivo configserver.yml:

```
spring:
  application:
    name: "configserver"  # Nome do serviço Config Server
  profiles:
    active: git  # Usando o repositório Git para armazenar as configurações
  cloud:
```

```
    config:
      server:
        git:
          uri: "https://github.com/Samuelrcf/eazybytes-microservices-config.git"  # URL do
repositório Git
          default-label: main  # Branch padrão
          timeout: 5  # Tempo limite para conexões
          clone-on-start: true  # Clona o repositório ao iniciar
          force-pull: true  # Força pull do repositório ao iniciar

management:
  endpoints:
    web:
      exposure:
        include: "*"  # Exposição de todos os endpoints do Actuator
  health:
    readiness-state:
      enabled: true  # Habilita a verificação de readiness
    liveness-state:
      enabled: true  # Habilita a verificação de liveness
  endpoint:
    health:
      probes:
        enabled: true  # Habilita probes de saúde

encrypt:
  key: "45D81EC1EF61DF9AD8D3E5BB397F9"  # Chave para criptografia de propriedades
sensíveis

server:
  port: 8083  # Porta do Config Server
```

Configuração do Eureka Server

Arquivo eureka.yml:

```
spring:
  application:
    name: "eurekaserver"  # Nome do serviço Eureka Server
  config:
    import: "optional:configserver:http://localhost:8083/"  # Conexão com o servidor de
configuração

management:
  endpoints:
```

```
  web:
    exposure:
      include: "*"  # Exposição de todos os endpoints do Actuator
health:
  readiness-state:
    enabled: true  # Habilita a verificação de readiness
  liveness-state:
    enabled: true  # Habilita a verificação de liveness
endpoint:
  health:
    probes:
      enabled: true  # Habilita probes de saúde
```

Dependências Necessárias

- **Eureka Server**: spring-cloud-starter-netflix-eureka-server
- **Eureka Client**: spring-cloud-starter-netflix-eureka-client
- **OpenFeign**: spring-cloud-starter-openfeign

Probes X States

Probes (Verificações Externas)

- **Definição**: As **probes** são verificações de saúde realizadas externamente, tipicamente por orquestradores de contêineres como o **Kubernetes**.
- **Objetivo**: Garantir que o serviço está pronto para receber tráfego (**readiness**) ou que está funcionando corretamente (**liveness**).
- **Configuração**: A configuração de probes é feita através de management.endpoint.health.probes.enabled.
- **Tipos de Probes**:
    - **Readiness Probe**: Verifica se a aplicação está pronta para receber tráfego. Uma resposta positiva indica que o serviço pode aceitar conexões, e o orquestrador pode rotear tráfego para ele.
    - **Liveness Probe**: Verifica se a aplicação ainda está viva e funcionando corretamente. Se essa verificação falhar, o orquestrador pode reiniciar a aplicação.
- **Uso em Orquestração**:
    - **Kubernetes** e outras ferramentas de orquestração de contêineres utilizam essas probes para garantir que o sistema esteja em boas condições de funcionamento antes de enviar tráfego ou enquanto o tráfego está sendo enviado.
- **Exemplo de Configuração**:

```
management:
  health:
    probes:
      enabled: true
```

States (Estados de Saúde Internos)

- **Definição**: **States** são indicadores internos da saúde da aplicação, como se ela está "viva" (em funcionamento) ou "pronta" (pronta para receber tráfego).
- **Objetivo**: A **readiness-state** e **liveness-state** indicam o status de prontidão e de funcionamento do sistema internamente, dentro do próprio Spring Boot.
- **Configuração**: As verificações internas de estado são configuradas em management.health.readiness-state.enabled e management.health.liveness-state.enabled.
- **States Externos**: Para que esses estados internos sejam acessíveis por orquestradores ou outros sistemas externos (como o Docker), é necessário expô-los através de endpoints de saúde do Spring Boot (exemplo: /actuator/health/readiness e /actuator/health/liveness).
- **Configuração de Exposição**: Para que as verificações de **readiness** e **liveness** estejam disponíveis externamente (por exemplo, para Kubernetes), você pode ativar as probes nas configurações de application.yml:

```
management:
  endpoints:
    web:
      exposure:
        include: "*"
  health:
    readiness-state:
      enabled: true  # Ativa o readiness-state
    liveness-state:
      enabled: true  # Ativa o liveness-state
  endpoint:
    health:
      probes:
        enabled: true  # Habilita probes de saúde
```

Diferença entre Probes e States

- **Probes** são verificações feitas por sistemas externos (como Kubernetes), que precisam acessar a saúde da aplicação.
- **States** são configurações internas mantidas pelo Spring Boot para determinar o status da aplicação.

Ambos são essenciais para garantir que a aplicação esteja pronta para atender aos clientes (readiness) e que ela continue funcionando conforme o esperado (liveness). Eles são configurados no Spring Boot Actuator para facilitar o monitoramento de saúde em ambientes de produção e orquestração.

Exposição das Probes e States Externamente

Para expor as probes e states para orquestradores externos, como **Kubernetes** ou **Docker**, é necessário garantir que esses endpoints estejam acessíveis. Com as configurações acima, ao acessar o endpoint /actuator/health/readiness, o orquestrador pode verificar a prontidão do serviço, e ao acessar /actuator/health/liveness, ele pode verificar a vivacidade da aplicação.

Exemplo de Exposição de Endpoints

- **Readiness**: http://localhost:8080/actuator/health/readiness
- **Liveness**: http://localhost:8080/actuator/health/liveness

Exemplo de Docker Compose

O arquivo **docker-compose.yml** é utilizado para configurar e orquestrar múltiplos containers Docker, facilitando a execução de serviços de microsserviços, como o **Config Server**, **Eureka Server**, entre outros. Aqui está um exemplo de como isso pode ser configurado:

```
services:
  configserver:
    image: "samuelrcf22/configserver:s8"
    container_name: configserver-ms
    ports:
      - "8083:8083"
    healthcheck:
      test: "curl --fail --silent localhost:8083/actuator/health/readiness | grep UP || exit 1" #
Realiza a verificação de saúde
      interval: 10s
      timeout: 5s
      retries: 10
      start_period: 10s
    extends:
      file: common-config.yml
      service: microservice-base-config

  eurekaserver:
    image: "samuelrcf22/eurekaserver:s8"
    container_name: eurekaserver-ms
    ports:
      - "8070:8070"
```

```
    depends_on:
      configserver:
        condition: service_healthy
    healthcheck:
      test: "curl --fail --silent localhost:8070/actuator/health/readiness | grep UP || exit 1"
      interval: 10s
      timeout: 5s
      retries: 10
      start_period: 10s
    extends:
      file: common-config.yml
      service: microservice-configserver-config
    environment:
      SPRING_APPLICATION_NAME: "eurekaserver" # Variáveis de ambiente para uso no
GitHub
```

Verificação de Saúde (Healthcheck) no Docker

- **Objetivo**: O Docker realiza verificações de saúde com base no **healthcheck** configurado. Neste exemplo, o Docker executa uma verificação de saúde usando **curl** para acessar o endpoint /actuator/health/readiness exposto pelo **Spring Boot**.
- **Funcionalidade**: Se a verificação de saúde falhar, o Docker pode realizar ações como reiniciar o contêiner.

Importante: No contexto de contêineres, **localhost** se refere à interface de rede interna do contêiner. Ou seja, o comando curl no **healthcheck** consulta o contêiner em si, não o host ou outros contêineres.

Problema de Não Manter Apenas um Ponto de Entrada

Ao utilizar múltiplos microsserviços, **manter apenas um ponto de entrada** para os microsserviços evita que os **clientes** precisem saber as **URLs e endpoints** de cada serviço. Isso centraliza o ponto de entrada e facilita o gerenciamento da comunicação entre os microsserviços, eliminando a necessidade de os clientes conhecerem a arquitetura interna.

- **Desafio**: Sem um ponto único de entrada, os clientes precisam conhecer a URL e os endpoints de todos os microsserviços, o que torna a manutenção difícil e a comunicação confusa.
- **Solução**: Usar um **API Gateway** para centralizar todas as requisições e fazer o roteamento entre os microsserviços.

API Gateway: Função e Roteamento

O **API Gateway** é um ponto centralizado de entrada que roteia as requisições para os microsserviços apropriados. Ele também pode lidar com questões como **balanceamento de**

carga e **autenticação**. No exemplo abaixo, ele redireciona as requisições de acordo com o caminho da URL:

- **Exemplo de Roteamento de Requisições**:
  - Antes do gateway: localhost:8080/api/create
  - Depois do gateway: localhost:8072/ACCOUNTS/api/create

O **API Gateway** atua como um ponto único de entrada e faz o roteamento para o **load balancer** do serviço desejado.

Configuração no Spring Boot (Java) para Roteamento

No Spring Boot, podemos configurar o roteamento das requisições diretamente na classe **Main** usando **RouteLocator** e **RouteLocatorBuilder**:

```
@Bean
RouteLocator srBankRouteConfig(RouteLocatorBuilder routeLocatorBuilder) {
   return routeLocatorBuilder.routes()
         .route(p -> p
               .path("/srbank/accounts/**")
               .filters(f -> f.rewritePath("/srbank/accounts/(?<segment>.*)","/${segment}"))
               .uri("lb://ACCOUNTS"))
         .route(p -> p
               .path("/srbank/cards/**")
               .filters(f -> f.rewritePath("/srbank/cards/(?<segment>.*)","/${segment}"))
               .uri("lb://CARDS"))
         .route(p -> p
               .path("/srbank/loans/**")
               .filters(f -> f.rewritePath("/srbank/loans/(?<segment>.*)","/${segment}"))
               .uri("lb://LOANS"))
         .build();
}
```

- **Explicação**:
  - A classe **RouteLocatorBuilder** permite definir as rotas de forma programática.
  - **rewritePath**: Essa função permite reescrever o caminho da URL para que ele corresponda à estrutura interna de cada microsserviço.
  - **lb://**: A configuração lb:// é usada para indicar que o gateway deve redirecionar a requisição para um **load balancer** que irá determinar qual instância do microsserviço será utilizada.

Observação Importante

- **Configuração com Java vs. YAML**:

- É recomendado configurar o roteamento com **Java**, pois ele oferece maior flexibilidade e controle em comparação com a configuração via **YAML**.
- Embora o **YAML** seja simples e direto para configurações básicas, **Java** permite uma personalização mais detalhada e suporte a configurações complexas, especialmente quando há limitações nas configurações padrão do Spring Cloud Gateway.

Uso de Filtros Globais em Aplicações Reativas

O **RequestTraceFilter** é um exemplo de filtro global utilizado em uma aplicação reativa baseada no Spring WebFlux, que é projetado para interceptar as requisições HTTP. O filtro verifica a presença de um **Correlation ID** nos cabeçalhos da requisição. Este **Correlation ID** é um identificador único associado a cada requisição, usado para rastrear a jornada da requisição por diferentes serviços dentro de uma arquitetura de microsserviços.

Objetivo do Correlation ID

- **Microsserviços**: O Correlation ID é fundamental para garantir que, mesmo em uma arquitetura distribuída, seja possível rastrear uma requisição enquanto ela passa por diferentes serviços. Ele ajuda a associar logs e eventos relacionados a uma mesma requisição, facilitando a depuração e análise de problemas.
- **Aplicação Monolítica**: Embora o uso do Correlation ID seja mais crítico em microsserviços, ele pode ser útil em monolitos para monitoramento, rastreamento de fluxos internos e geração de logs.

Funcionamento do RequestTraceFilter

O filtro funciona interceptando as requisições HTTP e verificando se o **Correlation ID** já está presente nos cabeçalhos. Se o ID estiver presente, ele é simplesmente registrado. Caso contrário, um novo ID é gerado e adicionado aos cabeçalhos da requisição. Isso garante que cada requisição tenha um identificador único, mesmo que o cliente (como o Postman) não o tenha fornecido.

```
HttpHeaders requestHeaders = exchange.getRequest().getHeaders();

if (isCorrelationIdPresent(requestHeaders)) {
   logger.debug("eazyBank-correlation-id found in RequestTraceFilter : {}",
filterUtility.getCorrelationId(requestHeaders));
} else {
   String correlationID = generateCorrelationId();
   exchange = filterUtility.setCorrelationId(exchange, correlationID);
   logger.debug("eazyBank-correlation-id generated in RequestTraceFilter : {}",
correlationID);
}
```

return chain.filter(exchange);

- **isCorrelationIdPresent**: Verifica se o Correlation ID já está presente nos cabeçalhos.
- **generateCorrelationId**: Se o Correlation ID não for encontrado, um novo é gerado.
- **setCorrelationId**: O novo Correlation ID gerado é adicionado aos cabeçalhos da requisição.

Este filtro permite que qualquer requisição que entre no sistema tenha um Correlation ID, independentemente de sua origem (cliente ou serviço). Isso facilita o rastreamento através dos microsserviços e também ajuda na consistência dos logs.

Mono<VOID> em Aplicações Reativas

O Spring WebFlux é projetado para lidar com processamento assíncrono e não bloqueante. Isso é alcançado através de **Mono** e **Flux**, que são tipos reativos que representam operações assíncronas. Eles permitem que o sistema responda rapidamente, sem bloquear as threads enquanto aguarda o resultado de uma operação (como uma consulta ao banco de dados ou uma chamada de rede).

- **Mono**: Representa uma operação que pode emitir **0 ou 1** item. É utilizado quando se espera uma resposta única ou nenhum valor. Um exemplo seria buscar um usuário específico em uma base de dados.
- **Flux**: Representa uma sequência de valores, podendo emitir **0 a muitos** itens. É ideal para fluxos de dados contínuos, como obter uma lista de usuários ou monitorar atualizações em tempo real.

Exemplos de Mono e Flux

- **Mono**: Para consultas que retornam um único item ou nenhum.

```
public Mono<User> findUserById(String id) {
   return userRepository.findById(id); // Retorna um único usuário
}
```

- **Flux**: Para consultas que retornam múltiplos itens ou um fluxo contínuo de dados.

```
public Flux<User> findAllUsers() {
   return userRepository.findAll(); // Retorna uma sequência de usuários
}
```

Por que usar Mono e Flux em vez de Tipos Tradicionais?

A principal vantagem de usar **Mono** e **Flux** é o processamento **não bloqueante** e **assíncrono**, permitindo uma melhor utilização dos recursos e a capacidade de lidar com grandes volumes de requisições simultâneas. Em vez de esperar bloqueado por uma operação (como uma consulta ao banco de dados), o **Mono** ou **Flux** devolve imediatamente um "contêiner" que

representará a operação assíncrona. Isso permite que outras operações sejam realizadas enquanto a operação esperada está em andamento.

- **Mono<User>**: Representa uma "promessa" que irá retornar um único **User** ou **nenhum** quando a operação assíncrona terminar.
- **Flux<User>**: Representa uma "promessa" que pode retornar uma sequência de **Users** ou nenhum item, ao longo do tempo.

Exemplo Comparativo

Método Tradicional (Bloqueante):

```
public User findUserById(String id) {
   return userRepository.findById(id); // Bloqueia até o resultado ser retornado
}
```

Neste caso, o método vai bloquear a thread até que o resultado seja obtido.

Método Reativo (Não Bloqueante):

```
public Mono<User> findUserById(String id) {
   return userRepository.findById(id); // Retorna uma "promessa" assíncrona
}
```

Aqui, **Mono** permite que o processamento continue enquanto o resultado é aguardado, liberando a thread para outras operações enquanto espera a resposta.

Resumo das Diferenças entre Mono e Flux

- **Mono**: Para uma única emissão de dados ou nenhuma. Ideal para operações que retornam um valor único, como buscas por ID.
- **Flux**: Para múltiplas emissões de dados. Ideal para operações que retornam uma sequência de itens ou um fluxo contínuo, como a obtenção de listas ou fluxos de dados em tempo real.

WebFlux x Mensageria

RabbitMQ e outros sistemas de mensageria como **Kafka**, **ActiveMQ**, e similares são usados para garantir que as mensagens sejam entregues mesmo que o consumidor esteja offline ou ocupado. Esses sistemas de mensageria persistem as mensagens nas filas até que sejam processadas, garantindo alta disponibilidade e confiabilidade, mesmo em cenários de falhas ou reinicializações do consumidor. A mensageria também pode fornecer garantias de entrega (ex. uma vez ou pelo menos uma vez) e persistência de mensagens.

Por outro lado, **Mono** e **Flux** no contexto de **Spring WebFlux** são tipos reativos usados para processar fluxos de dados de forma assíncrona e não bloqueante, mas **não têm persistência nativa**. Se ocorrer um erro ou se a aplicação for reiniciada enquanto uma mensagem está sendo processada, o dado pode ser perdido, pois esses fluxos de dados são gerenciados em memória. Portanto, se você precisar de garantias de persistência, como aquelas fornecidas pelos sistemas de mensageria, você precisará integrar o Spring WebFlux com sistemas de mensageria, como o RabbitMQ ou Kafka, para gerenciar filas e assegurar a entrega das mensagens.

ServerWebExchange no WebFlux

ServerWebExchange é uma abstração central no **Spring WebFlux**, representando o contexto completo de uma requisição HTTP durante seu processamento. Ele encapsula tanto a requisição quanto a resposta, permitindo que você acesse ou modifique ambos durante o ciclo de vida da requisição.

- **Requisição**: Contém informações como método HTTP (GET, POST), cabeçalhos, parâmetros, e corpo da requisição.
- **Resposta**: Representa os dados que serão enviados de volta ao cliente, incluindo cabeçalhos de resposta, status, e corpo.

A **ServerWebExchange** é particularmente útil para implementar filtros e manipuladores de requisição e resposta de forma reativa. O Spring WebFlux mantém o controle do contexto de ambos enquanto o processamento da requisição ocorre de forma assíncrona.

Fluxo Assíncrono com ServerWebExchange

1. **Recebendo a Requisição**: Quando uma requisição chega, o Spring cria um **ServerWebExchange** que contém todos os dados da requisição.
2. **Processamento Assíncrono**: A requisição pode ser processada de forma assíncrona, incluindo chamadas de banco de dados, interações com outros serviços, etc.
3. **Manipulação de Requisição e Resposta**: Durante o processamento, você pode modificar tanto os cabeçalhos da requisição quanto os da resposta. O fluxo é assíncrono, sem bloqueio das threads.
4. **Resposta Enviada**: Após o processamento, a resposta é enviada de volta ao cliente.

Exemplo de Global Filter para Manipulação de Cabeçalhos (Response Trace)

O exemplo abaixo mostra como um filtro global pode ser configurado para adicionar um **Correlation ID** ao cabeçalho de resposta após a solicitação ser processada e a resposta ser recebida de um microsserviço.

```
@Bean
GlobalFilter postGlobalFilter() {
   return (exchange, chain) -> {
      return chain.filter(exchange).then(Mono.fromRunnable(() -> {
```

```
            HttpHeaders requestHeaders = exchange.getRequest().getHeaders();
            String correlationId = filterUtility.getCorrelationId(requestHeaders);
            logger.debug("Updated the correlation id to the outbound headers: {}", correlationId);
            exchange.getResponse().getHeaders().add(filterUtility.CORRELATION_ID,
correlationId);
        }));
    };
}
```

Explicação do Código

- **chain.filter(exchange)**: O filtro continua o processamento da requisição, passando para o próximo filtro ou manipulador no pipeline.
- **Mono.fromRunnable()**: Após o processamento da requisição e recebimento da resposta, o **Mono** executa a ação definida no bloco Runnable, que, neste caso, adiciona o **Correlation ID** aos cabeçalhos da resposta.
- **exchange.getRequest().getHeaders()**: Obtém os cabeçalhos da requisição para recuperar o **Correlation ID**.
- **exchange.getResponse().getHeaders().add()**: Adiciona o **Correlation ID** aos cabeçalhos da resposta antes de enviá-la de volta ao cliente.

Logging no Gateway Server

Para ativar o log do **gateway server** e acompanhar o processamento de requisições no gateway, é possível configurar o nível de log no arquivo application.yml:

```
logging:
  level:
    com:
      eazybytes:
        gatewayserver: DEBUG
```

Essa configuração define que os logs para o pacote com.eazybytes.gatewayserver serão gerados no nível **DEBUG**, permitindo acompanhar em detalhes a execução e monitoramento do gateway.

Integração do WebFlux com Mensageria

Se você precisar integrar **Spring WebFlux** com sistemas de mensageria como RabbitMQ ou Kafka, pode usar o **Spring Integration** ou o **Spring Cloud Stream** para fornecer suporte a sistemas de mensagens assíncronas, garantindo persistência e entrega de mensagens, além de manter a natureza reativa da aplicação. Esses frameworks permitem o envio e recebimento de mensagens de forma não bloqueante, mas com as garantias de persistência e entrega típicas de sistemas de mensageria.

Resiliência em Microsserviços com Resilience4j

A resiliência é essencial em arquiteturas de microsserviços, dado o risco de falhas em cascata em sistemas distribuídos. **Resilience4j** oferece suporte a diversos padrões de resiliência, incluindo **Circuit Breaker**, **Retry**, e **Rate Limiter**, permitindo que os microsserviços lidem melhor com falhas e instabilidades.

Circuit Breaker Pattern

O **Circuit Breaker** ajuda a evitar falhas em cascata, protegendo os serviços dependentes de instabilidades.

Estados do Circuit Breaker:

1. **Closed (Fechado)**:
    a. As chamadas fluem normalmente.
    b. Se a taxa de erros ultrapassar o limite configurado, o estado muda para **Open**.
2. **Open (Aberto)**:
    a. Bloqueia todas as chamadas ao serviço problemático.
    b. Após um período configurado, muda para o estado **Half-Open** para testes.
3. **Half-Open (Meio-Aberto)**:
    a. Permite algumas chamadas de teste ao serviço.
    b. Se as chamadas tiverem sucesso, retorna ao estado **Closed**. Caso contrário, volta a **Open**.

Exemplo de Configuração no Arquivo application.yml:

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

Exemplo de Implementação com Gateway:

```
@Bean
RouteLocator srBankRouteConfig(RouteLocatorBuilder routeLocatorBuilder) {
   return routeLocatorBuilder.routes()
          .route(p -> p
                .path("/srbank/accounts/**")
                .filters(f -> f.rewritePath("/srbank/accounts/(?<segment>.*)", "/${segment}")
                      .addResponseHeader("X-Response-Time",
LocalDateTime.now().toString())
```

```
                .circuitBreaker(config -> config.setName("accountsCircuitBreaker")
                    .setFallbackUri("forward:/contactSupport")))
            .uri("lb://ACCOUNTS"));
}
```

- **Fallback**: Se o Circuit Breaker estiver aberto, as chamadas são redirecionadas para uma URI alternativa.

Configurações Prioritárias:

- Configurações específicas para uma rota ou microsserviço têm precedência sobre configurações globais.

Retry Pattern

O **Retry Pattern** tenta executar novamente operações que falham, acreditando que a falha pode ser transitória.

Conceitos Principais:

1. **Retry Logic**: Define o número de tentativas e as condições para repetir.
2. **Backoff Strategy**: Aumenta o tempo entre as tentativas para evitar sobrecarga.
3. **Idempotent Operations**: As operações devem ser idempotentes para evitar efeitos indesejados.

Configuração no Gateway Server (Rota de Cards):

```
.route(p -> p
   .path("/srbank/cards/**")
   .filters(f -> f
      .rewritePath("/srbank/cards/(?<segment>.*)", "/${segment}")
      .addResponseHeader("X-Response-Time", LocalDateTime.now().toString())
      .retry(retryConfig -> retryConfig
         .setRetries(3)
         .setMethods(HttpMethod.GET)
         .setBackoff(Duration.ofMillis(100), Duration.ofMillis(1000), 2, true)
      ))
   .uri("lb://CARDS"));
```

- **Configurações:**
    - **Retries**: 3 tentativas.
    - **Backoff Exponencial**: Intervalo inicial de 100ms, dobrando até 1s.
    - **Jitter**: Adiciona aleatoriedade ao intervalo para reduzir picos de carga.

Configuração no Microsserviço (application.yml):

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

Implementação no Microsserviço:

```
@Retry(name = "getBuildInfo", fallbackMethod = "getBuildInfoFallback")
@GetMapping("/build-info")
public ResponseEntity<String> getBuildInfo() {
   return ResponseEntity.status(HttpStatus.OK).body(buildVersion);
}

public ResponseEntity<String> getBuildInfoFallback(Throwable throwable) {
   return ResponseEntity.status(HttpStatus.OK).body("0.9");
}
```

Rate Limiter Pattern

O **Rate Limiter** controla o número de requisições permitidas em um intervalo de tempo, evitando sobrecargas e abusos.

Configuração no Gateway com Redis Rate Limiter:

O **RedisRateLimiter** é uma implementação de rate limiting que utiliza o Redis como backend para controle distribuído. Ele permite armazenar tokens e gerenciar limites de requisições, mesmo em arquiteturas distribuídas.

- **Configuração no Gateway:**

```
@Bean
RedisRateLimiter redisRateLimiter() {
   return new RedisRateLimiter(1, 1, 1); // 1 req/seg, 1 burst, 1 token
}

@Bean
KeyResolver userKeyResolver() {
```

```
    return exchange ->
Mono.justOrEmpty(exchange.getRequest().getHeaders().getFirst("user"))
            .defaultIfEmpty("anonymous");
}
```

- **Parâmetros do RedisRateLimiter:**
    - **Requisições por segundo (ReplenishRate)**: 1 requisição permitida por segundo.
    - **Burst Capacity**: 1 requisição extra permitida em curto período.
    - **Requested Tokens**: 1 token consumido por requisição.

Configuração do Redis no application.yml:

Para utilizar um container Redis com o Docker:

```
spring:
  data:
    redis:
      host: localhost
      port: 6379
      connect-timeout: 2s
      timeout: 1s
```

Testando o Rate Limiter com Apache Benchmark

O **Apache Benchmark (ab)** é usado para testar a capacidade de carga e validar configurações de rate limiting.

- **Comando de Teste:**

```
ab -n 10 -c 2 -v 3 http://localhost:8072/srbank/loans/api/contact-info
```

- **-n 10**: Total de 10 requisições.
- **-c 2**: Até 2 requisições simultâneas.
- **-v 3**: Mostra detalhes completos das requisições/respostas HTTP.

RateLimiter Annotation nos Microsserviços

A anotação @RateLimiter é usada diretamente nos métodos da API para limitar requisições.

- **Exemplo de Configuração no application.yml:**

```
resilience4j.ratelimiter:
  configs:
    default:
      timeoutDuration: 1000       # Tempo máximo de espera (em ms)
      limitRefreshPeriod: 5000    # Intervalo de renovação do limite (em ms)
```

```
    limitForPeriod: 1          # Máximo de 1 requisição a cada 5s
```

- **Exemplo de Implementação em um Método:**

```
@RateLimiter(name = "getProcessors", fallbackMethod = "getProcessorsFallback")
@GetMapping("/processors")
public ResponseEntity<String> getProcessors() {
   return
ResponseEntity.status(HttpStatus.OK).body(environment.getProperty("NUMBER_OF_PRO
CESSORS"));
}

public ResponseEntity<String> getProcessorsFallback(Throwable throwable) {
   return ResponseEntity.status(HttpStatus.OK).body("4");
}
```

Limitações:

- As configurações são globais para todos os métodos anotados. Não permitem personalização específica como no **RedisRateLimiter**.

Bulkhead Pattern

O **Bulkhead Pattern** é usado para isolar e proteger recursos do sistema de sobrecargas localizadas, garantindo que falhas em uma parte do sistema não afetem outras.

Tipos de Bulkhead:

1. **Semaphore Bulkhead (Baseado em Contadores):**

Limita o número de chamadas simultâneas a um recurso específico.

2. **Thread Pool Bulkhead (Baseado em Threads):**

Usa um pool separado de threads para processar requisições.

Configuração no application.yml:

```
resilience4j.bulkhead:
  configs:
    default:
      maxConcurrentCalls: 5       # Máximo de 5 chamadas simultâneas
      maxWaitDuration: 1000ms     # Tempo máximo para aguardar uma thread
```

Exemplo de Implementação:

```
@Bulkhead(name = "bulkheadExample", fallbackMethod = "fallbackMethod")
@GetMapping("/data")
public String fetchData() {
   return "Data processed successfully!";
```

```
}

public String fallbackMethod(Throwable throwable) {
   return "Service overloaded, try again later.";
}
```

Ordem de Execução dos Padrões no Resilience4j

Os padrões são executados em uma ordem específica para garantir a eficácia:

1. **Function**: O método principal.
2. **Bulkhead**: Limita as chamadas simultâneas.
3. **TimeLimiter**: Define um timeout para a execução.
4. **RateLimiter**: Controla o número de requisições.
5. **CircuitBreaker**: Abre ou fecha com base em falhas.
6. **Retry**: Tenta novamente em caso de falhas.

Configuração da Ordem no application.yml:

```
resilience4j:
  circuitbreaker:
    circuitBreakerAspectOrder: 1
  retry:
    retryAspectOrder: 2
```

- Valores maiores são executados **depois** de valores menores. Exemplo:
    - O Retry será executado antes do CircuitBreaker neste caso.
      Observabilidade e Monitoramento em Microsserviços

A **observabilidade** e o **monitoramento** são essenciais para manter a saúde e o desempenho de arquiteturas de microsserviços. A seguir, detalhamos os conceitos, práticas e ferramentas envolvidas.

Observabilidade x Monitoramento

- **Observabilidade**:
    - Foco em entender o estado interno do sistema por meio de sua saída.
    - Baseia-se em **três pilares**:
        - **Métricas**: Dados quantitativos sobre desempenho.
        - **Logs**: Detalhes sobre eventos no sistema.
        - **Traces**: Rastreiam requisições ponta a ponta.
    - Ferramentas: Prometheus, Grafana, Loki, Jaeger, Zipkin.
- **Monitoramento**:
    - Processo contínuo de coletar, analisar e alertar com base em métricas, logs e rastreamentos.

- Exemplo: Configurar alertas quando uso de CPU ultrapassa 80%.
- Ferramentas: Nagios, New Relic, Datadog.

Docker Compose para Observabilidade

O exemplo fornecido utiliza uma configuração robusta para um ambiente de observabilidade baseado em **Grafana**, **Loki**, **MinIO**, **NGINX** e **Alloy**.

Componentes Principais:

1. **Loki (Read/Write)**:
   a. Gerencia logs centralizados.
   b. Configurado em modos de leitura (read) e escrita (write) para escalabilidade.
   c. Requer configuração de dependências e volumes.
2. **MinIO**:
   a. Serviço de armazenamento usado como backend de dados para Loki.
   b. Configuração com credenciais seguras e suporte a métricas Prometheus.
3. **Grafana**:
   a. Visualização de métricas, logs e rastreamentos.
   b. Configuração com datasources e suporte anônimo habilitado.
4. **NGINX Gateway**:
   a. Proxy reverso para Loki, facilitando o roteamento de requisições.
5. **Alloy**:
   a. Agrega dados e oferece suporte para rastreamento distribuído.

Exemplo de Configuração docker-compose.yml

Trechos principais:

```
services:
  grafana:
    image: grafana/grafana:latest
    environment:
      - GF_PATHS_PROVISIONING=/etc/grafana/provisioning
      - GF_AUTH_ANONYMOUS_ENABLED=true
      - GF_AUTH_ANONYMOUS_ORG_ROLE=Admin
    depends_on:
      - gateway
    ports:
      - "3000:3000"
    healthcheck:
      test: ["CMD-SHELL", "wget --no-verbose --tries=1 --spider
http://localhost:3000/api/health || exit 1"]
      interval: 10s
      timeout: 5s
```

```
    retries: 5

  gateway:
    image: nginx:latest
    ports:
      - "3100:3100"
    entrypoint:
      - sh
      - -euc
      - |
        # Configuração dinâmica do NGINX
        cat <<EOF > /etc/nginx/nginx.conf
        server {
          listen 3100;
          location /api/prom/ {
            proxy_pass http://read:3100\$request_uri;
          }
        }
        EOF
        nginx -g "daemon off;"
```

Pilares da Observabilidade

1. **Métricas**:
    a. Coletadas com ferramentas como Prometheus.
    b. Exemplo: Taxa de erro HTTP 5xx, uso de CPU, ou latência.
2. **Logs**:
    a. Centralizados e gerenciados pelo Loki.
    b. Configuração de volumes no Loki:

```
volumes:
  - ../observability/loki/loki-config.yaml:/etc/loki/config.yaml
```

3. **Traces**:
    a. Ferramentas como **Jaeger** ou **Zipkin** podem ser integradas para rastreamento.
    b. O Alloy pode atuar como agregador de dados de rastreamento.

Exemplo Prático de Métricas com Prometheus e Grafana

1. **Configuração de Datasource no Grafana**:

```
datasources:
  - name: Loki
    type: loki
    url: http://gateway:3100
    access: proxy
```

2. **Visualização de Logs**:
    a. Acesse http://localhost:3000 (Grafana).
    b. Configure dashboards personalizados para exibir logs e métricas.

Iniciando o Ambiente

1. **Pré-requisitos**:
    a. Docker e Docker Compose instalados.
    b. Configurações adicionais no arquivo loki-config.yaml.
2. **Comandos**:
    a. Subir os serviços:

```
docker-compose up -d
```

    b. Verificar status:

```
docker-compose ps
```

3. **Acesso aos Serviços**:
    a. **Grafana**: http://localhost:3000
    b. **Loki** (logs): http://localhost:3100

Métricas e Monitoramento com Spring Boot Actuator, Micrometer, Prometheus e Grafana

A integração dessas ferramentas cria um pipeline poderoso para monitorar a saúde e o desempenho de microsserviços, desde a geração de métricas até a visualização.

1. Spring Boot Actuator

- **Propósito**: Prover endpoints para monitoramento e gerenciamento.
- **Funcionalidades**:
    - Exposição de métricas como saúde, memória, threads, etc.
    - Integração nativa com o **Micrometer**.
- **Exemplo de Endpoints**:
    - /actuator/health – Status geral da aplicação.
    - /actuator/metrics – Lista de métricas disponíveis.
    - /actuator/prometheus – Formato otimizado para coleta pelo Prometheus (necessário o Micrometer).
- **Configuração no application.yml**:

```
management:
  endpoints:
    web:
      exposure:
        include: health, metrics, prometheus
  health:
    livenessstate:
```

```
    enabled: true
  readinessstate:
    enabled: true
 metrics:
  tags:
    application: ${spring.application.name}
```

2. Micrometer

- **Propósito**: Middleware que traduz métricas do Actuator para o formato esperado por sistemas como Prometheus.
- **Configuração**:
    - Adicionar dependência Maven:

```
<dependency>
 <groupId>io.micrometer</groupId>
 <artifactId>micrometer-registry-prometheus</artifactId>
</dependency>
```

    - Ativar no Actuator:
        - O Micrometer automaticamente traduz as métricas em /actuator/prometheus.

3. Prometheus

- **Propósito**: Coletar, armazenar e consultar métricas geradas pelos microsserviços.
- **Configuração do Prometheus**:
    - Adicione os serviços no arquivo prometheus.yml:

```
global:
 scrape_interval: 5s
 evaluation_interval: 5s

scrape_configs:
 - job_name: 'accounts'
   metrics_path: '/actuator/prometheus'
   static_configs:
    - targets: ['accounts:8080']
 - job_name: 'loans'
   metrics_path: '/actuator/prometheus'
   static_configs:
    - targets: ['loans:8082']
```

    - Configuração via Docker Compose:

```
services:
 prometheus:
   image: prom/prometheus:v2.50.1
```

```
  container_name: prometheus
  ports:
    - "9090:9090"
  volumes:
    - ./prometheus.yml:/etc/prometheus/prometheus.yml
```

4. Grafana

- **Propósito**: Criar painéis visuais para análise de métricas.
- **Configuração com Prometheus**:
    - Após instalar o Grafana (via Docker ou manualmente), configure o **Prometheus como datasource**:
        - Vá para "Configuration" -> "Data Sources".
        - Adicione o URL do Prometheus (http://prometheus:9090).
    - Crie dashboards customizados para métricas como latência, uso de CPU e taxas de erro.

Tracing com OpenTelemetry

- **Propósito**: Rastrear a jornada de requisições através de diferentes microsserviços.
- **Configuração**:
    - Adicionar dependências OpenTelemetry:

```
<dependency>
  <groupId>io.opentelemetry</groupId>
  <artifactId>opentelemetry-api</artifactId>
</dependency>
<dependency>
  <groupId>io.opentelemetry</groupId>
  <artifactId>opentelemetry-sdk</artifactId>
</dependency>
```

    - Integre com visualizadores como **Jaeger** ou **Grafana Tempo**.

Pipeline Completo

1. **Loki**:
    a. Gerencia logs centralizados e rastreia eventos do sistema.
    b. Logs são exibidos no Grafana.
2. **Prometheus**:
    a. Coleta métricas de todos os microsserviços.
    b. Integração com Grafana para visualizações.
3. **Tempo/OpenTelemetry**:
    a. Para rastreamento distribuído.

Resumo das Integrações

Ferramenta	Função	Exemplo de Uso
Spring Boot Actuator	Fornece métricas iniciais	`/actuator/metrics`
Micrometer	Formata métricas	Conexão entre Actuator e Prometheus
Prometheus	Coleta e armazena métricas	Scrape de `/actuator/prometheus`
Loki	Armazena logs	Pesquisa e visualização de logs
Grafana	Dashboards e alertas	Painéis para métricas, logs e rastreamento
OpenTelemetry	Rastreamento distribuído	Rastrear requisições entre microsserviços

Comandos Finais

1. Subir o ambiente:

docker-compose up -d

2. Verificar serviços:
    a. Prometheus: http://localhost:9090
    b. Grafana: http://localhost:3000

Segurança em Microsserviços com OAuth2 e OpenID Connect

A segurança é um aspecto crucial na arquitetura de microsserviços, especialmente quando se trata de autenticação e autorização de usuários e serviços. O uso de **OAuth2** e **OpenID Connect (OIDC)** é uma solução robusta para lidar com esses desafios.

Principais Componentes do OAuth2

1. **Resource Owner (Proprietário do Recurso)**
    a. O usuário que possui os dados ou recursos protegidos.
    b. Exemplo: O proprietário de uma conta bancária acessada por um aplicativo de finanças.
2. **Client (Cliente)**
    a. A aplicação que deseja acessar os recursos em nome do Resource Owner.
    b. Exemplo: Um aplicativo de tarefas que precisa de acesso à lista de contatos do usuário.
3. **Authorization Server (Servidor de Autorização)**
    a. Responsável por autenticar o Resource Owner e emitir **tokens de acesso** para o Client.
    b. Exemplos: Keycloak, Auth0, Okta, ou servidores embutidos em plataformas (Google, Facebook).
4. **Resource Server (Servidor de Recursos)**
    a. Armazena os recursos protegidos e valida os tokens recebidos do Client.
    b. Exemplo: Uma API de contas bancárias que verifica o token antes de retornar informações sensíveis.

Scopes no OAuth2

Os **scopes** definem o nível de acesso que um Client terá aos recursos do Resource Owner.

- **Exemplo de Scopes:**
  - read_contacts: Permite ao Client apenas ler os contatos.
  - write_contacts: Permite adicionar ou modificar contatos.
- **Controle Granular**: Os Resource Servers validam os tokens e autorizam ações específicas com base nos scopes incluídos.

OpenID Connect (OIDC)

Enquanto o OAuth2 lida com autorização, o **OIDC** adiciona autenticação, fornecendo uma forma segura para que o Client verifique a identidade do Resource Owner.

1. **ID Token**
   a. Contém informações como nome, email e ID do usuário.
   b. Emitido pelo Authorization Server e usado pelo Client para identificar o usuário.
2. **Benefícios do OIDC**:
   a. **Autenticação Completa**: Ideal para Single Sign-On (SSO).
   b. **Simplificação**: Evita chamadas adicionais ao Resource Server para informações do usuário.
   c. **Compatibilidade com OAuth2**: Pode ser usado em conjunto com OAuth2 para um fluxo completo de autenticação e autorização.

Implementação com OAuth2 no Gateway Server

O Gateway Server é frequentemente o ponto de entrada em uma arquitetura de microsserviços. Configurar a segurança no Gateway é essencial para proteger os serviços downstream.

1. Configuração do SecurityConfig

A configuração de segurança usando o Spring Security e o WebFlux pode ser feita da seguinte forma:

```
@Configuration
@EnableWebFluxSecurity
public class SecurityConfig {

  @Bean
  SecurityWebFilterChain securityWebFilterChain(ServerHttpSecurity serverHttpSecurity) {
    serverHttpSecurity
      .authorizeExchange(exchanges -> exchanges
```

```
            .pathMatchers(HttpMethod.GET).permitAll()
            .pathMatchers("eazybank/accounts/**").authenticated()
            .pathMatchers("eazybank/cards/**").authenticated()
            .pathMatchers("eazybank/loans/**").authenticated()
        )
        .oauth2ResourceServer(oAuth2ResourceServerSpec ->
           oAuth2ResourceServerSpec.jwt(Customizer.withDefaults())
        )
        .csrf(csrfSpec -> csrfSpec.disable());
    return serverHttpSecurity.build();
  }
}
```

2. Configuração no application.yml

Configure o servidor de autorização (ex.: Keycloak) para validação dos tokens JWT:

```
security:
  oauth2:
    resourceserver:
      jwt:
        jwk-set-uri: "http://localhost:7080/realms/master/protocol/openid-connect/certs"
```

Aqui, o jwk-set-uri é usado para verificar os tokens JWT emitidos pelo Authorization Server.

Fluxo Completo com Keycloak

1. Configurar o Keycloak como Authorization Server

- Crie um Realm (ex.: master) no Keycloak.
- Adicione um Client (ex.: gateway-server) com:
    - Habilitado para autenticação via OAuth2.
    - Redirecionamentos configurados para o Gateway.

2. Obter um Token de Acesso

- Usuário autentica no Keycloak e o Authorization Server emite um **Access Token** (JWT).

3. Validar Tokens no Resource Server

- Resource Servers usam a URL configurada no jwk-set-uri para verificar a assinatura e validade dos tokens.

Construindo Microsserviços Orientados por Eventos

Microsserviços orientados por eventos oferecem escalabilidade, resiliência e flexibilidade em sistemas distribuídos, tornando-os ideais para cenários de alto desempenho e alta disponibilidade. Vamos explorar os conceitos e ferramentas para implementar essa abordagem.

Acoplamento Frouxo e Temporal

1. **Acoplamento Frouxo**
    a. Os serviços funcionam de forma independente, comunicando-se por meio de mensagens ou eventos.
    b. Exemplos:
        i. Uso de RabbitMQ ou Kafka para troca de mensagens.
        ii. Um produtor envia eventos sem precisar saber quem vai consumi-los.
2. **Acoplamento Temporal**
    a. Os serviços dependem da disponibilidade imediata um do outro para funcionar.
    b. Exemplo:
        i. Serviço A faz uma requisição HTTP para o Serviço B e precisa da resposta imediata para continuar.

Abordagens Imperativa vs. Reativa

Imperativa

- **Características:**
    - Comunicação síncrona e bloqueante.
    - Foco em chamadas diretas entre serviços.
- **Exemplo:**
    - Uma API REST que espera uma resposta imediata.

Reativa

- **Características:**
    - Comunicação assíncrona e não bloqueante.
    - Baseada em eventos ou mensagens.
- **Vantagens:**
    - Reduz acoplamento temporal.
    - Melhora a escalabilidade.
    - Permite aproveitar melhor os recursos do sistema.
- **Exemplo:**
    - Serviço publica eventos no Kafka, e consumidores processam as mensagens em tempo real.

Arquiteturas Orientadas por Eventos

1. **Event Streaming Model**
    a. Fluxos contínuos de eventos são armazenados e processados.
    b. Permite reprocessamento histórico.
    c. Ferramentas: **Apache Kafka**.
2. **Publisher/Subscriber Model**
    a. Eventos são enviados em tempo real para assinantes.
    b. Não armazena mensagens para acesso posterior.
    c. Ferramentas: **RabbitMQ**.

RabbitMQ no Contexto de Microsserviços

1. **Conceitos Básicos:**
    a. **Producer:** Envia mensagens.
    b. **Consumer:** Recebe mensagens.
    c. **Message Broker:** Facilita a comunicação entre produtor e consumidor.
    d. **Exchange:** Roteia mensagens para filas.
    e. **Queue:** Armazena mensagens até serem processadas.
2. **Tipos de Exchange:**
    a. **Direct:** Roteia mensagens com base em uma chave específica.
    b. **Fanout:** Envia mensagens para todas as filas vinculadas.
    c. **Topic:** Usa padrões na routing key para rotear mensagens.

Spring Cloud Functions e Stream

1. **Spring Cloud Functions**
    a. Facilita o desenvolvimento de funções reutilizáveis:
        i. **Supplier:** Produz dados.
        ii. **Consumer:** Consome dados.
        iii. **Function:** Processa dados.
2. **Spring Cloud Stream**
    a. Simplifica a integração com sistemas de mensagens.
    b. **Destination Binders:** Conecta o Spring Cloud Stream a sistemas como Kafka ou RabbitMQ.
    c. **Destination Bindings:** Lida com entradas e saídas no sistema de mensagens.
3. **StreamBridge**
    a. Envia mensagens para destinos definidos dinamicamente.
    b. Exemplo:

```
streamBridge.send("my-topic", message);
```

Kubernetes (K8s) no Suporte aos Microsserviços

O que é um Cluster?

- Conjunto de servidores que trabalham em conjunto para fornecer serviços confiáveis e escaláveis.

Componentes do Kubernetes:

1. **Nó Mestre (Master Node):**
    a. Gerencia o estado do cluster.
    b. Responsável por planejar e orquestrar os microsserviços.
2. **Nós de Trabalho (Worker Nodes):**
    a. Hospedam os microsserviços.
    b. Lidam com o tráfego e executam os contêineres.

Vantagens no Contexto de Microsserviços:

- **Escalabilidade:** Adiciona novos nós para lidar com maior carga.
- **Alta Disponibilidade:** Recria pods automaticamente em caso de falha.
- **Isolamento:** Cada microsserviço roda em um contêiner separado.

Resumo da Arquitetura Event-Driven

1. **Mensageria com RabbitMQ:**
    a. Abordagem publisher/subscriber.
    b. Uso de filas para desacoplamento.
2. **Event Streaming com Kafka:**
    a. Armazenamento de eventos para processamento histórico.
    b. Alta taxa de transferência para grandes volumes de dados.
3. **Spring Cloud Stream e Functions:**
    a. Abstração da lógica de comunicação com brokers.
    b. Conexão dinâmica com StreamBridge.
4. **Execução em Kubernetes:**
    a. Orquestração de microsserviços para escalabilidade e resiliência.