O Despertar do Guerreiro Java

A Arte Shinobi dos Microsserviços Apache Camel

Luis Zancanela

01

PREPARANDO O TERRENO DA BATALHA

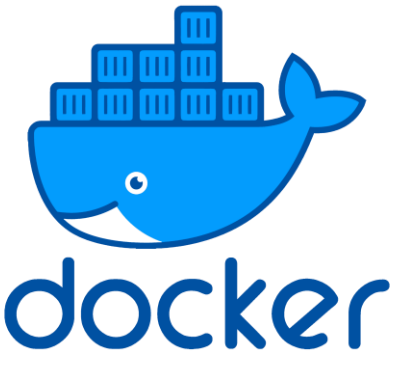
Preparando o Terreno da Batalha

Requisitos Básicos para Ingressar na Missão Shinobi

Antes de iniciar sua aventura, é imprescindível que seu arsenal esteja completo. Certifique-se de ter instalados os seguintes itens:

- JDK 21
- Maven
- Docker (para orquestrar componentes externos)

Além disso, sua missão integrará dois microsserviços que se comunicarão por meio de um message broker, utilizando o ActiveMQ via Docker Compose.



A FORÇA DO DRAGÃO DOCKER COMPOSE

A Força do Dragão Docker Compose

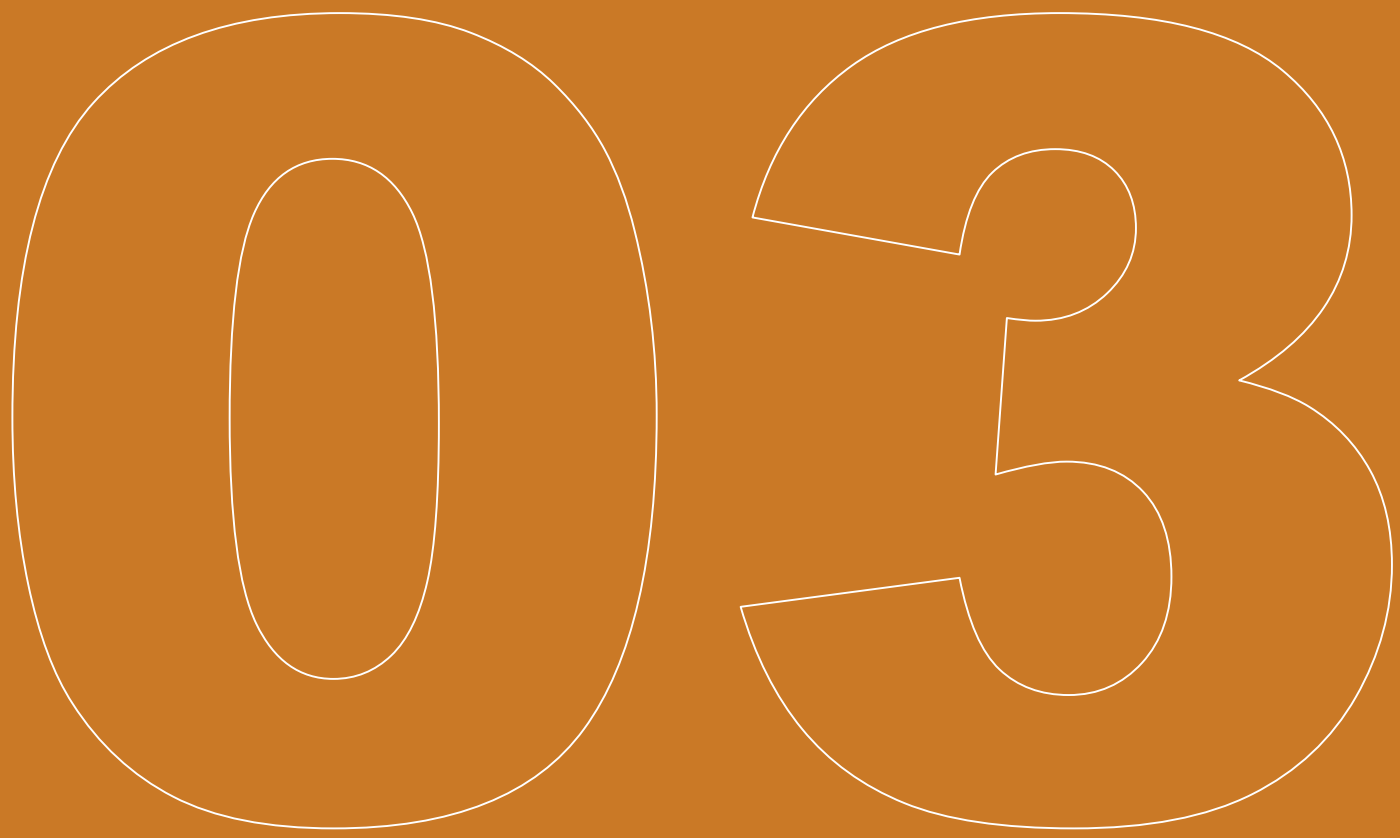
Configurando o ActiveMQ para a Comunicação Mística

Para que a comunicação entre nossos microsserviços ocorra com precisão, vamos utilizar o Docker Compose para lançar o ActiveMQ, nosso message broker lendário. Crie um arquivo chamado `docker-compose.yml` com o seguinte conteúdo:

RouteMicroservicoKage.java

```yaml
version: '3'
services:
  activemq:
    image: islandora/activemq:main
    ports:
      - "61616:61616"
      - "8161:8161"
    environment:
      - ACTIVEMQ_USER=admin
      - ACTIVEMQ_PASSWORD=admin
```

Este arquivo configura o ActiveMQ para escutar na porta 61616 (para mensagens) e 8161 (para interface administrativa). Assim, nosso clã de microsserviços terá um canal seguro para trocar informações.



FORJANDO O ARSENAL DO GUERREIRO

Forjando o Arsenal do Guerreiro

Criando Projetos Maven e Configurando o pom.xml

Para equipar cada microsserviço, crie dois projetos Maven independentes. No arquivo `pom.xml` de cada projeto, adicione as seguintes configurações mínimas:

pom.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.5.0</version>
        <relativePath/>
    </parent>

    <groupId>com.guerreiro.java</groupId>
    <artifactId>microservico-kage</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>microservico-kage</name>

    <properties>
        <java.version>21</java.version>
        <camel.version>4.10.5</camel.version>
    </properties>

    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.apache.camel.springboot</groupId>
                <artifactId>camel-spring-boot-dependencies</artifactId>
                <version>${camel.version}</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>
```

Forjando o Arsenal do Guerreiro

```
    <dependencies>
        <dependency>
            <groupId>org.apache.camel.springboot</groupId>
            <artifactId>camel-spring-boot-starter</artifactId>
        </dependency>
        <dependency>
            <groupId>org.apache.camel.springboot</groupId>
            <artifactId>camel-activemq-starter</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.apache.camel</groupId>
            <artifactId>camel-test-spring-junit5</artifactId>
            <version>${camel.version}</version>

</project>
```

Este **pom.xml** fornecerá os elementos necessários para que seus projetos possam implementar e administrar rotas com o Apache Camel, bem como integrar o ActiveMQ e funcionar com o Spring Boot.
É fundamental que cada microsserviço tenha seu próprio artifactId e nome. Neste exemplo, usaremos um microsserviço com artifactId de `microservico-kage` e outro denominado `microservico-shinobi`.

KAGE, O GUARDIÃO DOS PERGAMINHOS

Kage, O Guardião dos Pergaminhos

Validação do Pergaminho e Envio da Missão

O **Microsserviço Kage** lidera o clã secreto encarregado de receber os pergaminhos enviados pelo Imperador, que contêm as instruções para missões exclusivas do clã. Sua tarefa consiste em:

- Verificar se a primeira linha a mensagem é do Imperador
- Verificar se na segunda linha a mensagem é de fato para o Kage
- Obter o pedido da missão do pergaminho
- Enviar o pedido da missão para um Shinobi executar

Agora vamos ao laboratório do Kage para criar a lógica que validará o pergaminho secreto. Vamos fazer com implementação de um Processor do Apache Camel.

PergaminhoRecebidoProcessor.java

```java
@Component
public class PergaminhoRecebidoProcessor implements Processor {

    @Override
    public void process(Exchange exchange) throws Exception {
        String conteudo = exchange.getIn().getBody(String.class);
        String[] linhas = conteudo.split("\\r?\\n");

        validarCabecalho(linhas);

        String mensagem = extrairMensagem(linhas);
        exchange.getIn().setBody(mensagem);
    }

    private void validarCabecalho(String[] linhas) throws Exception {
        if (linhas.length < 3
                || !linhas[0].trim().contains("Imperador")
                || !linhas[1].trim().contains("Kage")) {
            throw new Exception("Arquivo inválido!");
        }
    }
```

Kage, O Guardião dos Pergaminhos

```
    private String extrairMensagem(String[] linhas) {
        StringBuilder mensagemBuilder = new StringBuilder();
        for (int i = 2; i < linhas.length; i++) {
            mensagemBuilder.append(linhas[i]).append("\n");
        }
        return mensagemBuilder.toString();
    }
}
```

Aqui finaliza a implementação do Processor, remetendo à tradição de escrita do Java e Spring Boot, imbuído do espírito da saga.

Agora, seguimos rumo à configuração da **rota do Apache Camel**, onde o Kage recebe o pergaminho e, com precisão, encaminha a missão secreta para que um Shinobi a execute com maestria.

RouteKage.java

```
@Component
public class RouteKage extends RouteBuilder {

    private final PergaminhoRecebidoProcessor pergaminhoRecebidoProcessor;

    public RouteKage(PergaminhoRecebidoProcessor pergaminhoRecebidoProcessor) {
        this.pergaminhoRecebidoProcessor = pergaminhoRecebidoProcessor;
    }

    @Override
    public void configure() {
        from("file://{{caminho.pergaminho.recebido}}?" +
                "noop=true" +
                "&include=.*\\.txt" +
                "&moveFailed={{caminho.pergaminho.erro}}" +
                "&delete=true")
                .routeId("receberPergaminho")
                .process(pergaminhoRecebidoProcessor)
                .to("activemq:queue:missoes")
                .log("Pergaminho lido e validado e missão enviada para a fila.");
    }
}
```

Kage, O Guardião dos Pergaminhos

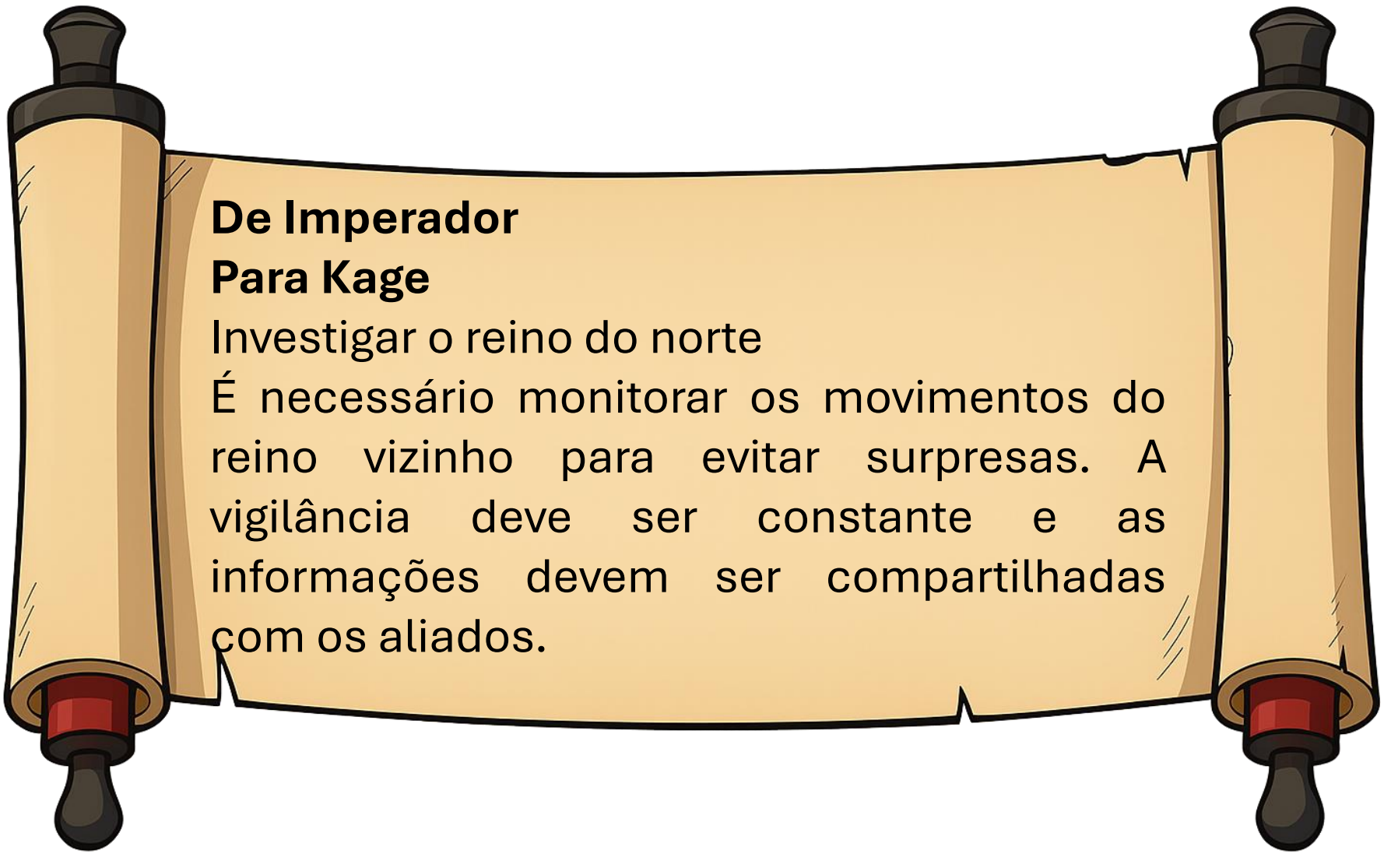
Para que o Kage estabeleça ao imperador o local de entrega do pergaminho e a rota para enviar a missão ultra secreta ao Shinobi, é necessário ajustar o arquivo `application.properties` do microsserviço Kage:

```
spring.application.name=microservico-kage

camel.springboot.main-run-controller=true

spring.activemq.broker-url=tcp://localhost:61616

caminho.pergaminho.recebido=../pergaminho-recebido
caminho.pergaminho.erro=../pergaminho-recebido/recusados
```



05

A MISSÃO SECRETA DO SHINOBI

A Missão Secreta do Shinobi

De prontidão para a missão!

Enquanto o Kage envia as mensagens, o **Microsserviço Shinobi** mantém-se alerta, escutando a fila `missoes`. Quando uma mensagem chega ele já pega da fila e registra que recebeu a missão. Eis exemplo prático da rota:

RouteShinobi.java

```java
@Component
public class RouteShinobi extends RouteBuilder {

    @Override
    public void configure() {
        from("activemq:queue:missoes")
                .routeId("receberMissao")
                .log("Nova missão recebida: ${body}");
    }
}
```

Não podemos esquecer do `application.properties`:

```
spring.application.name=microservico-shinobi

camel.springboot.main-run-controller=true

spring.activemq.broker-url=tcp://localhost:61616
```

UNIÃO DE FORÇAS O DESPERTAR FINAL

União de Forças
O Despertar Final

Integrando Sabedoria e Tecnologia para Vencer a Batalha Digital

Nesta aventura, dois microsserviços, o líder secreto (Microsserviço Kage) e o especialista de missões (Microsserviço Shinobi), unem forças por meio do **Apache Camel**, **Java**, **Spring Boot** e **ActiveMQ**. Com o **Docker Compose** orquestrando o cenário e o **Maven** gerenciando as dependências, você possui agora um exemplo prático e enxuto para criar sistemas integrados e escaláveis.

Esta aventura prática proporciona os passos mínimos para que você possa replicar e expandir o projeto. Use esses exemplos para expandir seu arsenal, testar novas rotas e, quem sabe, personalizar sua jornada na arte dos microsserviços. A cada linha de código, você se aproxima do domínio total dessa técnica ninja!

Referências e Links

Documentação Oficial Apache Camel:
https://camel.apache.org/camel-spring-boot

Repositório com implementação do exemplo:
https://github.com/didifive/ebook-com-ia-apache-camel-example

OBRIGADO POR LER!

Obrigado por Ler!

Agradeço por ter lido e aguardo seu feedback!

Esse ebook foi gerado com base em conteúdo textual e visual gerado por IA e revisado e diagramado por humano.

Os passos para desenvolvimento deste ebook estão no repositório GitHub abaixo (não esqueça de deixar sua estrela e me seguir por lá):

https://github.com/didifive/ebook-com-ia

Sobre o Autor

Olá, meu nome é Luis Zancanela e sou apaixonado por tecnologia e programação. Estou atuando na área de TI há mais de 18 anos, sendo mais de 3 anos como desenvolvedor backend. Abaixo, compartilho meu perfil no LinkedIn para que possamos nos conectar (sinta-se à vontade para me enviar seu feedback):

https://www.linkedin.com/in/luis-zancanela/

Se você ainda não conhecia, qual é a sua opinião sobre o Apache Camel na integração dos microsserviços?

Criado com paixão, dedicação e o compromisso sincero de compartilhar conhecimento 👦💻❤️🤝📚