Projeto de Microsserviços

Exercício 1: Implementação de um Microsserviço com Spring Boot

Descrição:

- Implemente um microsserviço em Spring Boot que gerencie uma lista de produtos. O microsserviço deve permitir operações CRUD (Create, Read, Update, Delete) através de uma API RESTful.

Visão Geral

Este projeto é uma arquitetura baseada em microsserviços projetada para fornecer serviços escaláveis e de fácil manutenção. Cada serviço é projetado para lidar com uma capacidade de negócio específica e pode ser desenvolvido, implantado e escalado de forma independente.

Serviços

O projeto consiste nos seguintes microsserviços:

- **Serviço de Produtos**: Gerencia uma lista de produtos, permitindo operações CRUD (Create, Read, Update, Delete) através de uma API RESTful.

Pré-requisitos

- Docker
- Java 11+
- Maven

Uso

Executando Localmente

1. Instale as dependências para o serviço de produtos:

   ```
   mvn clean install
   ```

2. Inicie o serviço:

   ```
   mvn spring-boot:run
   ```

3. Docker

```

```

```
docker run -d --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:3-management
```

API RESTful

O serviço de produtos permite as seguintes operações CRUD:

- **Criar Produto**:

```
POST /products
{
    "name": "Produto 1",
    "price": 100.0
}
```

- **Listar Produtos**:

```
GET /products
```

- **Atualizar Produto**:

```
PUT /products/{id}
{
    "name": "Produto Atualizado",
    "price": 150.0
}
```

- **Deletar Produto**:

```
DELETE /products/{id}
```

Documentação da API

A documentação da API é gerada automaticamente pelo Swagger e pode ser acessada em:

- http://localhost:8080/swagger-ui.html

Testes

1. Execute os testes para o serviço de produtos:

```
cd services/product-service
mvn test
```

Visualização dos Relatórios de Testes

Os relatórios de testes são gerados automaticamente e podem ser visualizados de duas formas:

1. **Relatórios HTML**:
    - Os relatórios HTML são gerados pelo Surefire Plugin e podem ser encontrados no diretório `target/surefire-reports/` após a execução dos testes.
    - Para visualizar os relatórios, abra os arquivos HTML no seu navegador.
2. **Relatórios de Cobertura de Código**:
    - Os relatórios de cobertura de código são gerados pelo Jacoco e podem ser encontrados no diretório `target/site/jacoco/` após a execução dos testes.
    - Para visualizar os relatórios, abra os arquivos HTML no seu navegador.
3. **Relatórios Publicados no GitHub Pages**:
    - Os relatórios de testes são publicados automaticamente no GitHub Pages após cada execução do pipeline de CI.
    - Acesse os relatórios através do link fornecido no repositório do GitHub.

Opcional

O **Act** é uma ferramenta para executar localmente workflows do GitHub Actions, permitindo que você teste pipelines sem a necessidade de fazer commits no repositório. Abaixo estão as etapas para usar o **Act** para testar o workflow especificado.

---

1. Instalar o Act

Siga as instruções abaixo para instalar o Act no seu ambiente local:

Para Linux/Mac:

```
brew install act
```

Para Windows (via Scoop):

```
scoop install act
```

Alternativamente, faça o download do executável:

- Acesse: Releases do Act
- Baixe e instale o binário adequado para o seu sistema operacional.

---

2. Configurar Dependências Locais

O Act utiliza imagens Docker para simular os runners do GitHub Actions. Verifique se você tem **Docker** instalado e configurado no seu ambiente.

Verificar Docker:

```
docker --version
```

Instalar Docker:

- Docker Desktop

---

3. Configurar o Act para seu Workflow

Passo 1: Verificar o Workflow

Certifique-se de que o arquivo do workflow (`.github/workflows/ci.yml`) está correto.

Passo 2: Configurar Secrets (opcional)

Se o workflow usa secrets, você precisará configurá-los localmente para o Act.

1. Crie o arquivo `.secrets` na raiz do repositório:

   ```
   GITHUB_TOKEN=my-github-token
   ```

   - Substitua `my-github-token` pelo valor real do token.
   - Outros secrets podem ser adicionados no mesmo formato.

2. Execute o Act referenciando o arquivo `.secrets`:

   ```
   act --secret-file .secrets
   ```

---

4. Executar o Workflow com Act

Passo 1: Simular o Workflow

Execute o workflow no ambiente local:

```
act push
```

- `push` simula o evento `on: push`.

Passo 2: Especificar a Imagem do Runner

Por padrão, o Act utiliza a imagem `nektos/act-environments-ubuntu:latest`. Para especificar outra imagem ou um runner mais leve:

```
act push -P ubuntu-latest=ghcr.io/catthehacker/ubuntu:act-latest
```

Passo 3: Executar Workflows Específicos

Para testar apenas um job específico, use:

```
act push -j build-test-report
```

Passo 4: Simular Outros Eventos

Você pode simular outros eventos, como `pull_request`:

```
act pull_request
```

---

5. Interpretar os Resultados

- O **Act** exibirá os logs do pipeline diretamente no console.
- Verifique se os passos estão sendo executados corretamente, como:
  - **Build do Maven:** `mvn clean install`
  - **Testes com Surefire:** `mvn test`
  - **Geração de relatórios Jacoco:** Verifique se os relatórios estão na pasta `target/site/jacoco`.

---

6. Limitações e Notas

1. **Actions de Terceiros:**

- Certifique-se de que as actions usadas no workflow (`actions/checkout@v2`, `actions/setup-java@v2`, etc.) são suportadas pelo Act.
- Caso contrário, substitua por uma alternativa local ou simule o comportamento.

2. **Secrets no Local:**
   - Use um arquivo `.secrets` para configurar variáveis sensíveis.
   - O Act não armazena secrets de forma segura.

3. **Volumes Locais:**
   - O Act mapeia diretórios locais no Docker, garantindo que arquivos gerados durante o pipeline estejam disponíveis após a execução.

Exemplo: Executar o Workflow do CI

```
# Simula um push para a branch main
act push

# Simula um evento pull_request
act pull_request

# Executa um job específico
act push -j build-test-report

# Especifica uma imagem personalizada para simular o runner
act push -P ubuntu-latest=ghcr.io/catthehacker/ubuntu:act-latest
```

Com o **Act**, você pode iterar e testar localmente seus workflows antes de fazer push para o repositório. Isso economiza tempo e reduz o ciclo de desenvolvimento. Se precisar de mais ajuda para configurar ou debugar seus workflows, estarei à disposição!

Exercício 4: Aplicação dos Princípios SOLID Descrição:

- Revise o código do microsserviço desenvolvido nos exercícios anteriores e refatore-o para garantir que ele esteja aderente aos princípios SOLID.
- Adicione comentários explicando as modificações feitas e como elas melhoram o código em termos de design e manutenibilidade.
- Single Responsibility Principle (SRP) Antes: ProdutoService lidava com operações de CRUD, mapeamento entre entidades e DTOs, e comunicação com RabbitMQ. Depois: Cada responsabilidade foi separada: ProdutoMapper: Responsável apenas pelo mapeamento entre entidades e DTOs. MessagingService: Responsável apenas pela comunicação com RabbitMQ. ProdutoService: Foca apenas na lógica de negócios relacionada aos produtos, delegando outras responsabilidades.

- Open/Closed Principle (OCP) Antes: Qualquer modificação nas funcionalidades de mapeamento ou mensagens exigia mudanças na classe ProdutoService. Depois: As classes estão abertas para extensão (poderão ser adicionadas novas implementações de ProdutoMapper ou MessagingService), mas fechadas para modificações, pois não necessitam alterar a ProdutoService.

- Liskov Substitution Principle (LSP) Implementações concretas (ProdutoMapperImpl, RabbitMQService) podem substituir suas respectivas interfaces sem alterar o comportamento esperado do sistema.

- Interface Segregation Principle (ISP) Interfaces específicas (ProdutoMapper, MessagingService) são definidas para evitar que classes clientes dependam de interfaces que não utilizam.

- Dependency Inversion Principle (DIP) ProdutoService depende de abstrações (ProdutoMapper, MessagingService), não de implementações concretas (ProdutoMapperImpl, RabbitMQService). Facilita a substituição de implementações, melhora a testabilidade (facilitando o uso de mocks), e promove um acoplamento fraco entre os componentes.