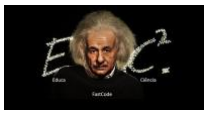


Boas Práticas de Desenvolvimento de Microsserviços com Java 17

O desenvolvimento de microsserviços tornou-se um padrão amplamente adotado para construir sistemas distribuídos, escaláveis e resilientes. Com a abordagem de microsserviços, cada serviço é desenvolvido de forma independente, encapsulando funcionalidades específicas, o que facilita a manutenção e a escalabilidade. Neste artigo, exploraremos as boas práticas técnicas para o desenvolvimento de microsserviços utilizando **Java 17**, com exemplos didáticos que ilustram a aplicação desses conceitos.

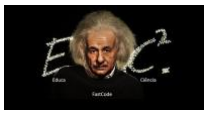
1. Arquitetura Baseada em Domínio (Domain-Driven Design - DDD)

No desenvolvimento de microsserviços, é fundamental que cada serviço reflita um **bounded context** (contexto delimitado) de um domínio específico. A ideia é que cada microsserviço encapsule completamente as regras de negócio de seu domínio, o que garante alta coesão dentro do serviço e baixo acoplamento entre os serviços.

Exemplo Didático

Em uma aplicação de e-commerce, temos três domínios principais: **Pedidos**, **Pagamentos** e **Clientes**. Cada domínio pode ser implementado como um microsserviço independente:

```
@RestController
@RequestMapping("/pedidos")
public class PedidoController {

    @Autowired
    private PedidoService pedidoService;

    @PostMapping
    public ResponseEntity<PedidoDTO> criarPedido(@RequestBody PedidoDTO pedidoDTO) {
        PedidoDTO novoPedido = pedidoService.criarPedido(pedidoDTO);
        return new ResponseEntity<>(novoPedido, HttpStatus.CREATED);
    }

    // Demais endpoints de CRUD
}
```



Boas Práticas

- **Separação de Domínios**: Cada microsserviço deve corresponder a um domínio específico, de forma que não haja sobreposição de responsabilidades.
- **Isolamento Total**: A lógica de negócio, banco de dados e regras de validação devem estar encapsulados em cada serviço, evitando dependências diretas entre microsserviços.

2. Comunicação entre Microsserviços com APIs Restful

Microsserviços podem se comunicar de forma síncrona usando **APIs RESTful** ou de forma assíncrona usando mensageria com ferramentas como **Apache Kafka** ou **RabbitMQ**. Java 17 introduz o HttpClient, uma API moderna e mais eficiente para realizar requisições HTTP.

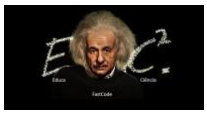
Exemplo Didático - Comunicação Síncrona via HTTP

A seguir, temos um exemplo de comunicação entre o microsserviço de **Pagamentos** e o de **Pedidos**:

```
public class PedidoClient {

    private final HttpClient client = HttpClient.newHttpClient();

    public PedidoDTO buscarPedidoPorId(String pedidoId) throws IOException,
InterruptedException {
        HttpRequest request = HttpRequest.newBuilder()
            .uri(URI.create("http://localhost:8080/pedidos/" + pedidoId))
            .GET()
            .build();

        HttpResponse<String> response = client.send(request,
HttpResponse.BodyHandlers.ofString());
        return new ObjectMapper().readValue(response.body(), PedidoDTO.class);
    }
}
```

Boas Práticas

- **Idempotência**: As APIs devem ser idempotentes para garantir que múltiplas requisições produzam o mesmo efeito. Isso é crucial em sistemas distribuídos para evitar inconsistências.
- **Timeout e Retry**: Configure tempos de timeout e implemente tentativas automáticas de repetição (retry) para gerenciar falhas em chamadas entre serviços.



3. Resiliência e Tolerância a Falhas

Em sistemas distribuídos, falhas são inevitáveis. Para garantir a resiliência dos microsserviços, padrões como **Circuit Breaker**, **Bulkhead** e **Rate Limiting** devem ser implementados. **Resilience4j** é uma biblioteca popular no ecossistema Java para aplicar essas práticas.

Exemplo Didático - Circuit Breaker com Resilience4j

```
@CircuitBreaker(name = "pedidoService", fallbackMethod = "fallbackBuscarPedido")
public PedidoDTO buscarPedidoPorId(String pedidoId) {
    return pedidoClient.buscarPedidoPorId(pedidoId);
}

public PedidoDTO fallbackBuscarPedido(String pedidoId, Throwable t) {
    // Retorna um objeto de fallback em caso de falha
    return new PedidoDTO(pedidoId, "Fallback", "Detalhes indisponíveis");
}
```

Boas Práticas

- **Circuit Breaker**: Use Circuit Breaker para evitar que falhas em cascata afetem outros serviços.
- **Retries**: Ao reexecutar chamadas falhas, aplique backoff exponencial para evitar sobrecarregar serviços que estão temporariamente inativos.

4. Versionamento de APIs

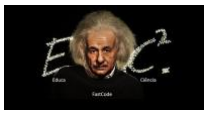
Conforme as APIs evoluem, é importante garantir que as novas versões não quebrem os clientes existentes. O versionamento pode ser implementado via URL ou cabeçalhos HTTP.

Exemplo Didático

```
@RestController
@RequestMapping("/v1/pedidos")
public class PedidoControllerV1 {
    // Lógica para a versão 1
}

@RestController
@RequestMapping("/v2/pedidos")
public class PedidoControllerV2 {
    // Nova lógica para a versão 2
}
```

Boas Práticas

- **Backward Compatibility**: Mantenha compatibilidade retroativa sempre que possível.



- **Depreciação Gradual**: Informe aos clientes com antecedência sobre a descontinuação de versões antigas, permitindo tempo para migrações.

5. Observabilidade: Logs, Monitoramento e Tracing

Observabilidade é a capacidade de monitorar e depurar microsserviços em produção. Ferramentas como **Spring Boot Actuator**, **Prometheus** e **Grafana** fornecem monitoramento, enquanto **Jaeger** e **Zipkin** ajudam no tracing distribuído.

Exemplo Didático - Spring Boot Actuator

```
// application.properties
management.endpoints.web.exposure.include=health,info

// Exemplo de Health Check
@RestController
public class HealthCheckController {

    @GetMapping("/health")
    public String checkHealth() {
        return "OK";
    }
}
```

Boas Práticas

- **Distributed Tracing**: Utilize ferramentas de tracing distribuído para mapear requisições em vários serviços.
- **Logs Estruturados**: Adote padrões de logs centralizados para facilitar a auditoria e depuração.

Conclusão

Microsserviços trazem uma arquitetura flexível, mas exigem disciplina no design e na implementação para que sua escalabilidade e resiliência sejam atingidas. Com o suporte de **Java 17**, ferramentas modernas como **Spring Boot**, **Resilience4j** e tecnologias de monitoramento e tracing, é possível construir sistemas distribuídos robustos e fáceis de manter.

EducaCiência FastCode para a comunidade

Referências

- Resilience4j Documentation
- Spring Boot Actuator Documentation
- Jaeger - OpenTracing
- Java 17 HttpClient API