



**INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DA BAHIA
CAMPUS - SANTO ANTÔNIO DE JESUS - BAHIA**

**CURSO DE TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE
SISITEMAS**

MARCELO DE JESUS

RONALDO CORREIA

Simulação de Microservices Resilientes com Aplicação de Padrões Arquiteturais e Diagnóstico de Anti-patterns

MARCELO DE JESUS

RONALDO CORREIA

Simulação de Microservices Resilientes com Aplicação de Padrões Arquiteturais e Diagnóstico de Anti-patterns

Relatório técnico da
atividade de Padrões de Projeto-
Simulação de Microservices
Resilientes com Aplicação de
Padrões Arquiteturais e
Diagnóstico de Anti-patterns ,
elaborado como requisito parcial
de avaliação para a disciplina
de Padrões de Projeto pelo Prof.
Felipe Silva.

SANTO ANTONIO DE JESUS – BA

2025

1. INTRODUÇÃO

O presente relatório apresenta de forma descritiva a implementação de um sistema de Usuário e Pedido, cujo objetivo foi proporcionar na prática a aplicação de padrões arquiteturais modernos em cenários de microservices. O intuito foi desenvolver um protótipo distribuído, onde foi organizado em duas etapas, versão inicial e versão refatorada. A versão inicial, teria como objetivo apresentar falhas evidentes arquiteturais, como chamadas diretas entre serviços, sem controle de falhas, alto acoplamento entre módulos, e violações explícitas de princípios SOLID, por exemplo, é notório a presença de serviços centralizados em únicas classes com múltiplas responsabilidades (God Service) dentre outras violações como, Spaghetti Services, princípios SRP (Single Responsibility Principle) e DIP (Dependency Inversion Principle).

Na versão refatorada foram aplicadas melhorias arquiteturais que trouxeram maior organização e resiliência ao sistema. O API Gateway foi implementado para centralizar o acesso externo e reduzir o acoplamento entre serviços; a Inversão de Controle (IoC) foi utilizada para facilitar a injeção de dependências e modularizar a comunicação; e foram previstas extensões para inclusão dos padrões Circuit Breaker e Bulkhead.

Assim, o processo realizado permitiu comparar uma arquitetura distribuída frágil e acoplada com uma versão mais modular, resiliente e alinhada a boas práticas arquiteturais, proporcionando aprendizado tanto técnico quanto conceitual sobre os desafios e soluções em microservices.

2. DIAGNOSTICO DE ANTI-PATTERNS

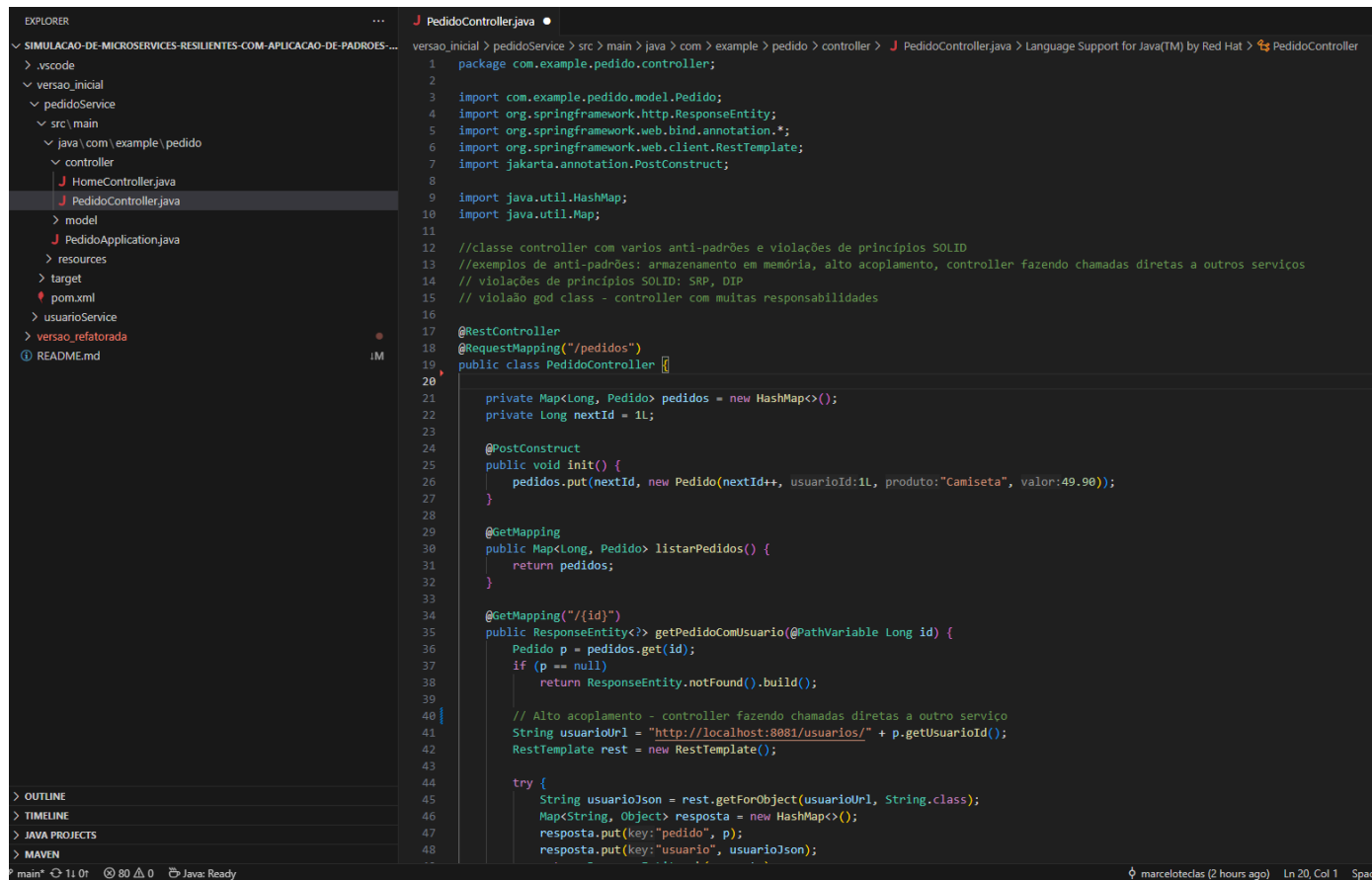
2.1 Justificativa:

Como abordado na introdução, o desenvolvimento da atividade ficou dividida em duas etapas, onde a versão inicial foi criada sem a preocupação de uso dos padrões arquiteturais nem tao pouco dos princípios SOLID. Durante a primeira etapa da atividade, ao desenvolver a versão inicial, foram identificados anti-patterns, que impactaram diretamente a clareza, manutenibilidade e a resiliência do sistema. Segue abaixo alguns dos problemas identificados nesta primeira versão:

- **God Services:** Uma das problemáticas mais evidentes foi a concentração de inúmeras responsabilidades em determinados serviços/classes. Na implementação inicial, a classe de Pedido assumia lógicas e responsabilidades que deveriam ser desacopladas da mesma e direcionada para outros setores, evitando que este modulo acumule múltiplas funções, tornando a mesma um ponto crítico de falhas e dificultando a evolução independente de componentes específicos.
- **Spagetti-Services:** Outro anti-pattern evidente foi a presença muito grande de dependências entre serviços, com chamadas diretas e acoplamento forte. O microservice de Pedido dependia de forma notória a implementação interna do microserviço de Usuário, o que trazia dificuldade na substituição, evolução ou simulação de falhas em um dos módulos. De acordo com a análise acima, essa arquitetura traz características do anti-patterns Spaghetti Services, onde a comunicação se torna desorganizada entre os serviços.

Esse foi alguns dos problemas identificados na versão inicial, de apenas duas classes de serviços desenvolvidas. Este diagnostico serviu como ponto de partida para refatoração guiada, implementada na versão posterior. A introdução do API Gateway e da Injeção de Dependências (IoC) teve como objetivo central a mitigação desses problemas como: Reduzir o acoplamento, melhorar a modularidade e preparar o sistema para aplicação de padrões de resiliência como o Circuit Breaker e Bulkhead.

2.2 Classe com problemas detectado:



The screenshot shows a VS Code editor with a project named "SIMULACAO-DE-MICROSERVICES-RESILIENTES-COM-APLICACAO-DE-PADROES-...". The file explorer on the left shows the project structure, with the file "PedidoController.java" selected in the "controller" directory. The main editor displays the code for "PedidoController.java".

The code includes the following comments indicating violations of SOLID principles:

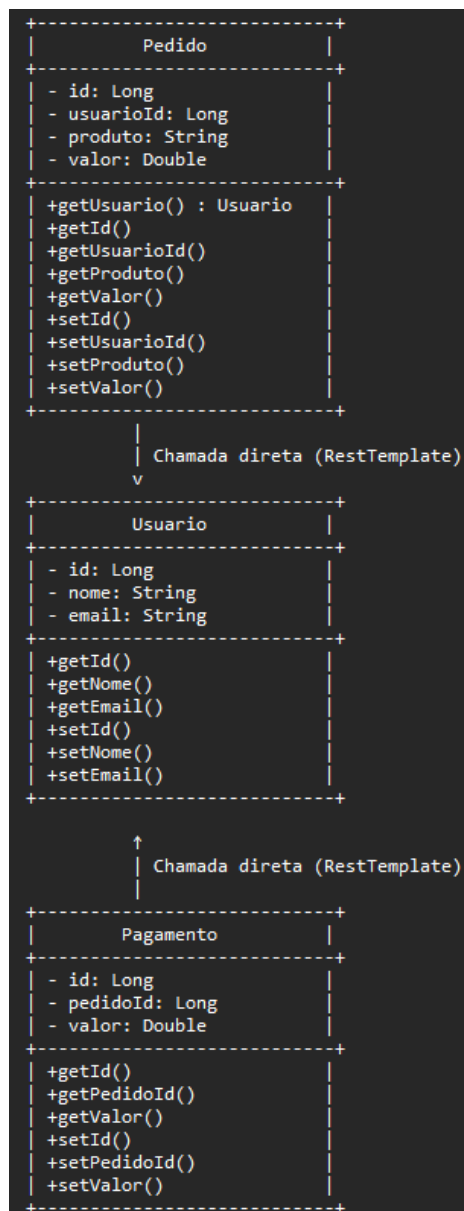
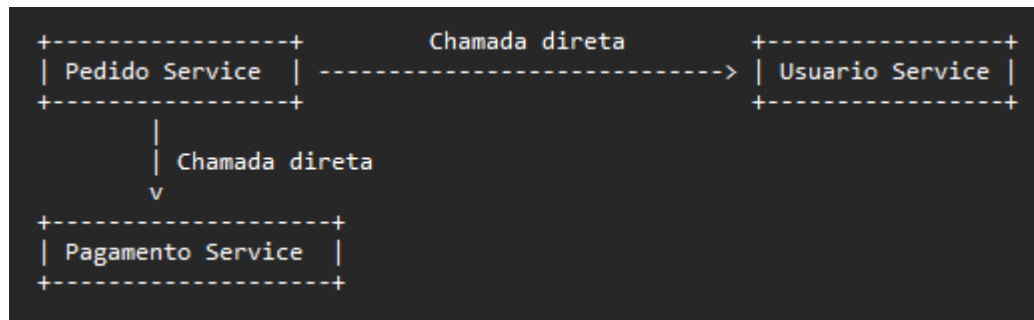
- Line 12: `//classe controller com varios anti-padrones e violacoes de principios SOLID`
- Line 13: `//exemplos de anti-padrones: armazenamento em memoria, alto acoplamento, controller fazendo chamadas diretas a outros servicos`
- Line 14: `// violacoes de principios SOLID: SRP, DIP`
- Line 15: `// violação god class - controller com muitas responsabilidades`

The code defines a `PedidoController` class with the following methods:

- `init()`: Initializes a `HashMap` of `Pedidos` and sets `nextId` to 1.
- `listarPedidos()`: Returns the `pedidos` map.
- `getPedidoComUsuario(@PathVariable Long id)`: Retrieves a `Pedido` by ID. If not found, it returns `ResponseEntity.notFound().build()`.
- `try` block: Makes a REST call to `http://localhost:8081/usuarios/` + `p.getId()` to retrieve user information and updates the `pedido` in the `resposta` map.

The status bar at the bottom indicates the file is named "main*", the cursor is at line 1, column 0, and the Java IDE is ready.

2.3 UML Inicial – Microservice com Anti-Patterns:



Explicação: O pedido service depende diretamente do Usuário Service usando o Rest Template, criando dessa forma um alto acoplamento, exemplo clássico do anti-pattern **Spaghetti Service**.

3. VIOLAÇÕES DO SOLID

3.1 Justificativa

Na implementação inicial do protótipo de microservices, foram identificadas diversas violações aos princípios SOLID, especialmente nos controllers `UsuarioController` e `PedidoController`. Segue abaixo as principais violações do SOLID:

- Violação do Princípio da Responsabilidade Única (SRP – Single Responsibility Principle): O `PedidoController` assume múltiplas responsabilidades: gerencia o armazenamento dos pedidos, expõe endpoints REST e orquestra chamadas para o `UsuarioService`. A responsabilidade de gerenciar pedidos deveria estar isolada em uma camada de serviço ou repositório, enquanto o controller deveria apenas coordenar as requisições HTTP.
- Violação do Princípio da Inversão de Dependência (DIP – Dependency Inversion Principle): A dependência direta do `PedidoController` em `RestTemplate` e na URL fixa do `UsuarioService` evidencia alto acoplamento. O controller depende de uma implementação concreta para acessar outro serviço, o que dificulta a testabilidade e impede a flexibilidade da arquitetura, contrariando o DIP, que preconiza que módulos de alto nível não devem depender de módulos de baixo nível, mas sim de abstrações.
- Violação do Princípio Aberto/Fechado (OCP – Open/Closed Principle): O `UsuarioController` armazena os usuários em memória (`HashMap`) diretamente no controller. Caso seja necessário alterar a forma de persistência (por exemplo, migrar para um banco de dados relacional), seria necessário modificar diretamente o controller, contrariando o OCP, que orienta que classes devem estar abertas para extensão, mas fechadas para modificação.
- God Service / Controller Multifuncional: O `PedidoController` concentra lógica de negócio, acesso a dados e integração externa, configurando um típico God Service, onde uma única classe assume demasiadas responsabilidades, tornando o sistema mais frágil, difícil de testar e de manter.
- Violação do Princípio da Segregação de Interfaces (ISP – Interface Segregation Principle)

Embora não haja interfaces explícitas no código, a ausência delas já representa um problema. Os controllers consomem diretamente estruturas de dados e serviços externos sem abstração, o que impede a criação de contratos específicos para cada tipo de operação. Isso dificulta a evolução do sistema e a reutilização de componentes.

Exemplo: O PedidoController consome diretamente o usuario-service via RestTemplate, sem interface intermediária que permita abstrair ou simular esse comportamento em testes.

- Violação do Princípio da Substituição de Liskov (LSP – Liskov Substitution Principle)
Embora não haja herança explícita entre controllers, o uso de estruturas concretas como HashMap para persistência em memória dentro dos controllers impede a substituição por implementações mais robustas (como repositórios JPA) sem modificar o comportamento da classe. Isso fere o espírito do LSP, que preconiza que componentes devem ser substituíveis sem efeitos colaterais.
Exemplo: A substituição do HashMap por um repositório real exigiria alterações diretas no UsuarioController, quebrando o encapsulamento.

Essas violações refletem problemas comuns em microservices que não fazem uma análise previa e segue as boas práticas de arquiteturas atuais, demonstrando a necessidade de refatoração utilizando padrões arquiteturais modernos.

4. COMPARATIVO ENTRE ARQUITETURA INICIAL E FINAL

Aspecto	Versão Inicial	Versão Refatorada
Acoplamento	Alto (chamada direta via URL)	Baixo (via abstração e IoC)
Resiliência	Inexistente	Circuit Breaker + Bulkhead
Manutenção	Difícil	Facilitada por modularização
Testabilidade	Baixa	Alta (mock de dependências)
Escalabilidade	Limitada	Pronta para escalar horizontalmente

5. REFLEXÃO CRÍTICA

Em arquiteturas distribuídas reais, falhas são inevitáveis — mas não precisam ser catastróficas. A aplicação de padrões como Circuit Breaker e Bulkhead permite que os serviços se protejam contra falhas em cascata e sobrecarga. Além disso, práticas como monitoramento contínuo (logs, métricas e alertas), timeouts e retries bem calibrados, e fallbacks inteligentes são essenciais para garantir disponibilidade e experiência do usuário.

O isolamento de recursos com Bulkhead, o uso de filas assíncronas e a limitação de concorrência ajudam a preservar a estabilidade do sistema mesmo sob carga. Por fim, testes de resiliência com ferramentas como Chaos Monkey simulam falhas reais e ajudam a validar a robustez da arquitetura.

6. EXPLICAÇÃO TÉCNICA DA APLICAÇÃO DOS PADRÕES

O projeto gatewayService aplica dois padrões de resiliência amplamente utilizados em sistemas distribuídos:

1. Circuit Breaker (Resilience4j)

- **Objetivo:** Evitar chamadas repetidas a serviços instáveis.
- **Como foi aplicado:**
 - Anotação `@CircuitBreaker(name = "usuarioService", fallbackMethod = "fallbackUsuario")`
 - Quando o serviço de usuários falha consecutivamente, o circuito é "aberto" e o método de fallback é acionado.
- **Benefício:** Reduz o tempo de resposta e protege o sistema contra sobrecarga.

2. Bulkhead (Resilience4j)

- **Objetivo:** Isolar recursos para evitar que a falha de um serviço afete os demais.
- **Como foi aplicado:**
 - Anotação `@Bulkhead(name = "usuarioBulkhead", type = Bulkhead.Type.THREADPOOL)`
 - Define limites de concorrência para chamadas ao serviço de usuários.
- **Benefício:** Garante que outros serviços continuem funcionando mesmo sob alta carga ou falha de um deles.

3. Fallback

- **Objetivo:** Fornecer uma resposta alternativa em caso de falha.
- **Como foi aplicado:**
 - Métodos `fallbackUsuario` e `fallbackPedido` retornam mensagens padronizadas quando os serviços estão indisponíveis.
- **Benefício:** Melhora a experiência do usuário e evita erros genéricos.

4. LINK REPOSITORIO

- <https://github.com/marcelotecnica/Simulacao-de-Microservices-Resilientes-com-Aplicacao-de-Padrees-Arquiteturais-Antipatterns.git>

5. REFERÊNCIAS

- FOWLER, Martin. Microservices. Disponível em: <https://martinfowler.com/articles/microservices.html>. Acesso em: 07 set. 2025.
- MARTIN, Robert C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Upper Saddle River: Prentice Hall, 2017.
- NEWMAN, Sam. Building Microservices: Designing Fine-Grained Systems. 2. ed. O'Reilly, 2019.
- TURNER, D.; BUNCE, M. Anti-patterns in Microservice Architecture. IEEE Software, v. 37, n. 1, p. 45-53, 2020.
- SPRING. Dependency Injection. Disponível em: <https://spring.io/guides/gs/consuming-rest/>. Acesso em: 07 set. 2025.