

# Relatório Projeto de Tolerância a Falhas

Alexandre Dantas dos Santos

Antonio Higino Bisneto Leite Medeiros

Ignacio Saglio Rossini

Novembro de 2025

# Chapter 1

## Introdução

### 1.1 Apresentação

O documento aqui descrito visa apresentar o relatório de desenvolvimento do projeto de tolerância a falhas, requisito de avaliação obrigatória da disciplina: Tópicos Especiais em Engenharia de Software IV, ministrada pelo professor Dr. Gibeon Soares de Aquino Junior.

### 1.2 Visão Geral

Este projeto implementa um sistema distribuído de compra de passagens aéreas, desenvolvido com foco em tolerância a falhas. A solução adota uma arquitetura de microsserviços, na qual os componentes se comunicam por meio de serviços REST.

### 1.3 Características Principais

- Projeto desenvolvido na linguagem de programação Java (versão: 21);
- Arquitetura de microsserviços desenvolvida com Spring Boot;
- Comunicação entre serviços por meio de APIs REST;
- Containerização e orquestração utilizando Docker.

## Chapter 2

# Arquitetura e Desenvolvimento

### 2.1 Características

De forma resumida, temos a seguinte visão geral do sistema:

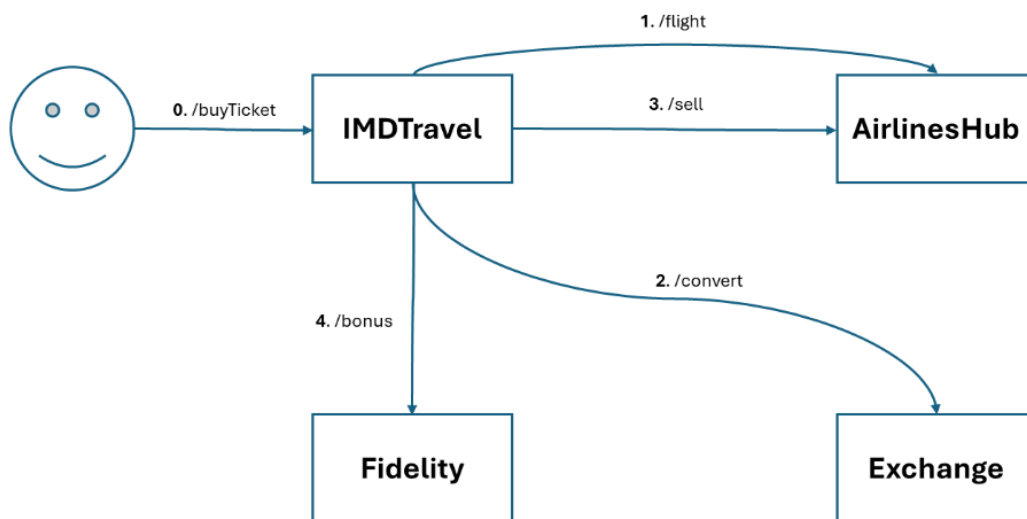


Figure 2.1: Visão geral do sistema

O ImdTravel atua como o ponto central de entrada da aplicação, sendo responsável por orquestrar todo o processo de compra. Ele recebe a requisição do usuário e coordena as chamadas aos demais microserviços, garantindo que cada etapa do processo seja executada corretamente.

O microserviço AirlinesHub é responsável por manipular todas as operações relacionadas ao voo. Ele provê informações sobre voos disponíveis e realiza o processamento da venda, entregando um identificador único da transação. Suas operações são essenciais para validar disponibilidade e registrar a compra.

O serviço Exchange fornece a taxa de conversão de moeda utilizada na compra. Ele é consultado sempre que é necessário calcular valores convertidos, garantindo que o sistema opere com informações atualizadas de câmbio.

Por fim, o microserviço Fidelity é responsável pelo programa de fidelidade. Ele registra os pontos de bônus dos usuários conforme as transações realizadas.

## 2.2 Serviços

A solução é composta por um conjunto de microsserviços independentes, cada um responsável por uma parte específica do fluxo de compra de passagens. Esses serviços comunicam-se por meio de chamadas HTTP e seguem um modelo desacoplado, permitindo escalabilidade, flexibilidade e facilidade de manutenção. A seguir, são detalhados os principais componentes da arquitetura e suas funcionalidades.

### 1. **ImdTravel** (/buyTicket)

- Atua como ponto central de entrada do sistema.
- Coordena e gerencia a comunicação entre os microsserviços.
- **Endpoint:**
  - **POST** /buyTicket: Processa compra com parâmetros:
    - \* **flight**: número do voo a ser comprado
    - \* **day**: data do voo a ser comprado
    - \* **user**: ID do usuário que está executando a compra
- Porta: 8081

### 2. **AirlinesHub** (/flight, /sell)

- Gerencia compra do voo.
- **Endpoints:**
  - **GET** /flight: Retorna dados do voo com os parâmetros.
  - **POST** /sell: Processa venda e retorna ID único da transação.
  - **Parâmetros (para ambos os endpoints):**
    - \* **flight**: número do voo a ser comprado
    - \* **day**: data do voo a ser comprado
- Porta: 8084

### 3. **Exchange** (/convert)

- Fornece taxa de conversão de moeda.
- **Endpoint:**
  - **GET** /exchange: Retorna taxa de conversão (número real positivo)
- Porta: 8083

### 4. **Fidelity** (/bonus)

- Gerencia programa de fidelidade.
- **Endpoint:**
  - **POST** /bonus: Registra pontos de bônus com parâmetros:
    - \* **user**: ID do usuário
    - \* **bonus**: valor inteiro do bônus
- Porta: 8082

O projeto é executado sem um banco de dados real. Ainda assim, alguns dados manipulados durante a execução são mantidos em estruturas de dados armazenadas em memória, como as informações dos vôos disponíveis (AirlinesHub) e o registro dos bônus dos usuários (Fidelity). Essas informações são gerenciadas pelas classes do projeto FlightService e FidelityService. Como exemplo se houver uma consulta de um vôo que não exista na "base de dados", será retornado para o usuário uma resposta do tipo: 204 (No Content).

## 2.3 Implementação

O projeto foi desenvolvido em Java e estruturado como um ambiente distribuído baseado em microserviços. Cada módulo corresponde a um serviço independente, implementado como um projeto Java isolado, permitindo modularidade, escalabilidade e a simulação realista de cenários de tolerância a falhas.

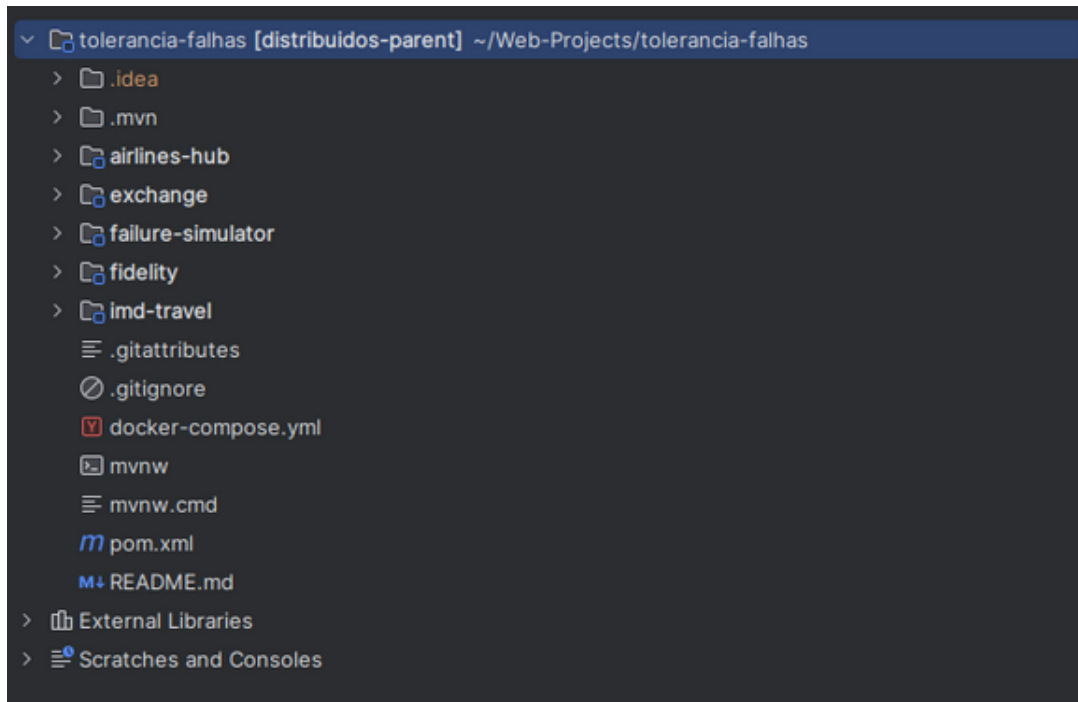


Figure 2.2: Visão geral do projeto

De forma resumida, temos:

- **distribuidos-parent:** Este é o módulo pai (parent) do projeto multimodular. Nenhuma lógica de negócio é executada aqui. É apenas a base central do build.
- **airlines-hub:** Este módulo representa o serviço de companhias aéreas, responsável por fornecer dados dos vôos e processar as vendas.
- **exchange:** Este módulo implementa o serviço de câmbio, que gera uma taxa de conversão de câmbio no endpoint.
- **fidelity:** Este módulo fornece o serviço de programa de fidelidade e responsável por registrar bônus para o usuário.
- **imd-travel:** Este é o serviço orquestrador. Representa o gateway da compra e executa o fluxo completo.
- **failure-simulator:** Este módulo é o coração do cenário de falhas. Ele é responsável por definir tipos de falhas que serão simuladas nos microserviços.

## Chapter 3

# Falhas e estratégias de tolerância a falhas

### 3.1 Falhas

Nesta fase foram implementadas quatro falhas simuladas seguindo o padrão (tipo de falha, probabilidade, duração). As falhas são gerenciadas pelo módulo `failure-simulator`, que atua como orquestrador central das falhas.

De forma geral, as falhas implementadas seguem o seguinte escopo, obedecendo o padrão técnico: Fail (Type, Probability, Duration):

- Request 1: Fail (Omission, 0.2, 0s)
- Request 2: Fail (Error, 0.1, 5s)
- Request 3: Fail (Time = 5s, 0.1, 10s)
- Request 4: Fail (Crash, 0.02, )

Todas as falhas são orquestradas pelo módulo `failure-simulator`. Esse módulo contém as especificações (`FailureSpec`) com os campos `type`, `probability` e `durationSeconds`, bem como o estado de falhas ativas (`ActiveFailure`). A decisão de aplicar uma falha para um endpoint é tomada pelo `FailureManager` com base nessas specs.

**OmissionFailure:** A falha de omissão é implementada usando `DeferredResult` no lado do controller. Quando o `FailureManager` decide omitir uma resposta para um endpoint, o fluxo retorna um `DeferredResult` que nunca é completado (nem `setResult` nem `setErrorResult`). Dessa forma o Spring mantém a conexão HTTP aberta e o cliente fica aguardando até estourar o timeout do cliente.

**ErrorFailure:** A falha de erro retorna imediatamente uma resposta HTTP 500. Foi implementada uma estratégia `ErrorFailure` que implementa a interface `FailureStrategy` e devolve: `ResponseEntity.status(500).build()`.

**TimeFailure:** A falha de tempo é simulada pelo `FailureManager` registrando uma `ActiveFailure` com duração de 10 segundos. Durante esse período, para cada requisição ao endpoint afetado, o código aplica um `Thread.sleep(5000)` (delay de 5s) antes de continuar o processamento ou responder. O efeito é que todas as requisições recebidas enquanto a falha estiver ativa sofrerão um atraso de aproximadamente 5 segundos cada.

**CrashFailure:** A falha de crash é representada pelo `FailureManager` sinalizando que o endpoint deve morrer. Na fase de simulação essa semântica foi implementada de forma direta: quando sorteada, o sistema executa uma parada do processo (por exemplo, `Runtime.getRuntime().halt()`).

## 3.2 Tolerância a falhas

Os seguintes requisitos foram definidos para a implementação dos mecanismos de tolerância a falhas em cada endpoint do sistema:

- Falha no Request 1 : Estratégia: Livre escolha.
- Falha no Request 2 : Estratégia: Em caso de erro, deve-se usar o valor de taxa como sendo a média das últimas 10 taxas.
- Falha no Request 3 : Estratégia: Erros de latência na resposta (maior que 2s), devem ser consideradas como erro, impedindo a continuidade da operação de venda e falhando graciosamente para o agente externo. Desenvolver proteções para que a alta latência no serviço AirlinesHub não comprometa significativamente o desempenho do IMDTravel e nem o tempo de resposta para o agente externo.
- Falha no Request 4 : Estratégia: Não impedir a venda. Permitir processar a bonificação quando o subsistema voltar ao normal.

No endpoint responsável por retornar os dados do voo (/flight), foram aplicadas três técnicas de tolerância a falhas: timeout (5 segundos), retry (até 3 tentativas) e fallback com uso de cache local. Em resumo, o funcionamento ocorre da seguinte forma:

Caso a resposta ultrapasse o limite de 5 segundos, é lançada a exceção personalizada `FlightServiceInternalException`, que aciona automaticamente o mecanismo de retry, limitado a três tentativas. Se, mesmo após essas tentativas, o serviço ainda falhar, entra em ação o fallback, que recupera os dados a partir de um cache local contendo o último valor armazenado com sucesso para aquele voo e para aquela data específicos (ou seja, para a chave `flight + day`).

A seguir, são apresentados os detalhes da estrutura desse mecanismo, junto a classe: `FlightService.java`:

```
public Map<String, Object> getFlight(Long flight, String day, boolean ft) { 1 usage  Alexandre Danta

    if (!ft) {
        Map<String, Object> resp = getFlightData(flight, day, ft);
        flightCache.put(flight + "_" + day, resp); // Atualiza o cache
        return resp;
    }

    int attempts = 0;

    while (attempts < 3) {
        try {
            Map<String, Object> resp = getFlightData(flight, day, ft);
            logger.warn("Tentativa " + (attempts + 1) + " bem-sucedida.");
            flightCache.put(flight + "_" + day, resp); // Atualiza o cache
            return resp;
        } catch (ResourceAccessException | FlightServiceInternalException e) {
            attempts++;
            logger.warn("[FT] Tentativa " + attempts + " falhou. Motivo: " + e.getMessage());
        }
    }

    // Fallback em cache
    logger.info("[FT] Todas as tentativas falharam. Verificando cache...");
    Map<String, Object> cached = flightCache.get(flight + "_" + day);

    if (cached != null) {
        logger.info("[FT] Cache encontrado. Retornando fallback.");
        return cached;
    }

    throw new RuntimeException("Falha por omissão - sem cache disponivel.");
}
```

Figure 3.1: Trecho de código de `FlightService.java`

Como pode se observar no código, a cada requisição bem-sucedida ao serviço de voos, os dados retornados são armazenados no cache interno representado pelo *map* `flightCache`. Esse cache utiliza como chave a combinação (`flight + day`), garantindo que sempre seja salvo e posteriormente recuperado o último valor correspondente àquele voo e àquela data. Assim, em cenários de falha, o sistema pode recorrer ao valor previamente armazenado para a mesma consulta, oferecendo um mecanismo eficiente de fallback.

Além disso, destaca-se a variável `attempts`, responsável por controlar o número de tentativas realizadas durante o processo de `retry`. Esse contador é incrementado a cada falha nas requisições, permitindo que o sistema limite o processo a no máximo três tentativas antes de acionar o fallback. Tal estrutura garante maior robustez ao endpoint, reduzindo impactos de lentidão ou indisponibilidade temporária do serviço externo.

No endpoint responsável por retornar a taxa de conversão de câmbio (`/convert`), foi aplicada a técnica de fallback com média das últimas 10 taxas com uso de taxa padrão. O funcionamento ocorre da seguinte forma:

Caso o mecanismo de tolerância a falhas esteja ativo, o sistema armazena o valor da taxa de conversão em uma pilha com capacidade para até 10 elementos. Isso permite manter um histórico recente de valores válidos. Em situações de exceção, como na indisponibilidade do serviço externo, o sistema aciona o fallback, recuperando como valor de retorno a média dos elementos atualmente armazenados na pilha. Ainda assim, caso não existam valores presentes na pilha, é retornado um valor padrão: 5.0.

A seguir temos os detalhes desta implementação, junto a classe: `ExchangeService.java`:

```
public Double getRate(boolean ft) { 1 usage  3 pulisaglio+1

    try {
        Double rate = rest.getForObject(url: exchangeUrl + "/convert", Double.class);

        storeRate(rate);
        return rate;
    } catch (Exception e) {

        if (!ft) {
            throw e;
        }

        Double fallback = averageLastRates();

        if (fallback == null) {
            logger.warn("[FT] Falha e histórico vazio. Usando taxa padrão: {}", DEFAULT_RATE);
            return DEFAULT_RATE;
        }

        logger.warn("[FT] Falha na requisição. Usando média dos últimos {} valores: {}", lastRates.size(), fallback);

        return fallback;
    }
}

private void storeRate(Double rate) { 1 usage  3 pulisaglio

    if (rate == null) return;

    if (lastRates.size() == 10) {
        lastRates.removeFirst();
    }

    lastRates.addLast(rate);
}
```

Figure 3.2: Trecho de código de `ExchangeService.java`

Como pode se observar no código, a cada requisição bem-sucedida é realizada a chamada do método: `storeRate()` que processa o armazenamento da taxa recuperada. No caso da exceção, é recuperado o valor médio dos valores, através da variável `fallback` em conjunto com o método: `averageLastRates()`.

No endpoint responsável por retornar a taxa de conversão de câmbio (/sell), foi aplicada a técnica de timeout curto (2 segundos), com disparo de falha graciosa. O funcionamento ocorre da seguinte forma:

Com o mecanismo de tolerância a falhas ativado, caso o tempo de resposta exceda 2 segundos, uma RuntimeException é lançada, registrando em log o motivo da falha na venda, neste caso, a latência superior ao limite estabelecido.

A seguir segue o detalhamento desta implementação junto a classe: SalesService.java:

```
public Long registerSale(Long flight, String day, boolean ft) { 1 usage 2 pulisaglio +1

    RestTemplate rest = defaultRest;

    if (ft) {
        SimpleClientHttpRequestFactory factory = new SimpleClientHttpRequestFactory();
        factory.setConnectTimeout(2000);
        factory.setReadTimeout(2000);
        rest = new RestTemplate(factory);
        logger.info("[FT] Tolerância ativa: timeout de 2s configurado no Request 3");
    }

    try {
        ResponseEntity<Long> resp = rest.postForEntity(
            String.format("%s/sell?flight=%s&day=%s", airlinesUrl, flight, day),
            request, null,
            Long.class
        );

        if (!resp.getStatusCode().is2xxSuccessful() || resp.getBody() == null) {
            throw new NoSuchElementException("Venda não registrada.");
        }

        return resp.getBody();
    } catch (HttpClientErrorException.BadRequest e) {
        throw new IllegalArgumentException("Parâmetros inválidos.", e);
    } catch (HttpServerErrorException e) {
        throw new RuntimeException("Erro no serviço Airlines ao registrar venda.", e);
    } catch (ResourceAccessException e) {
        if (ft) {
            logger.warn("Venda não registrada devido à latência (>2s). Cancelando venda...");
            throw new RuntimeException("Venda cancelada por latência (>2s).", e);
        }
        throw e;
    }
}
```

Figure 3.3: Trecho de código de SalesService.java

Já no endpoint responsável por processar a pontuação de fidelidade do usuário (/fidelity), foi aplicada a técnica de fallback por fila em memória com retry agendado. O funcionamento ocorre da seguinte forma:

Caso haja algum erro na transmissão haverá o armazenamento dos dados usuário e seu bônus correspondente junto a uma fila. Neste ponto, a cada 30 segundos haverá uma nova tentativa de registro desse bônus de forma assíncrona.

A seguir segue o detalhamento desta implementação junto a classe: FidelityService.java:

```
    } catch (Exception e) {
        logger.error("[Fidelity] Falha no envio imediato → user={}, bonus={}, erro={}",
            user, bonus, e.getMessage());

        if(!ft) throw e;

        queue.add(new FidelityTask(user, bonus, ft: true));

        logger.warn("[Fidelity] Task adicionada à fila como fallback → user={}, bonus={}",
            user, bonus);
    }
}

/**
 * Executada a cada 30s.
 * Processa tasks pendentes que falharam no envio imediato.
 */
@Scheduled(fixedDelay = 30000) no usages & pulisaglio
public void processQueue() {
    if (queue.isEmpty()) {
        logger.debug("[Fidelity] Fila vazia, nada a processar.");
        return;
    }

    logger.info("[Fidelity] Processando fila pendente (tamanho={})", queue.size());

    FidelityTask task;
    while ((task = queue.poll()) != null) {

        if (!task.ft()) {
            logger.error("[Fidelity] ERRO: Task encontrada na fila com ft=false → user={}, bonus={}",
                task.user(), task.bonus());
            // descarta a task
            continue;
        }
    }
}
```

Figure 3.4: Trecho de código de FidelityService.java

Como é observado, em caso de problemas na transmissão, é realizada a adição do registro pendente a fila, no trecho: `queue.add(new FidelityTask(user, bonus, true))`, afim de garantir uma nova transmissão futura. Na imagem é observado também o trecho do método: `processQueue()`, que tenta realizar o processo de reenvio do bônus pendente. O tempo de cada solicitação é programado usando o comando `@Scheduled`, com valor fixo de 300000 (30 segundos).

### 3.3 Análise das limitações

Seguem algumas limitações identificadas no projeto desenvolvido:

- **Ausência de circuit breaker:** O sistema continua tentando acessar serviços instáveis mesmo após sua degradação, o que pode aumentar o tempo de resposta e sobrecarregar ainda mais o serviço remoto. Esse mecanismo poderia ser implementado utilizando a biblioteca *Resilience4j*.
- **Cache local não distribuído:** O cache dos dados é armazenado apenas em memória local, o que implica perda de estado caso a instância atual seja reiniciada ou apresente falha. Uma solução mais robusta seria utilizar um banco de dados, como *MongoDB*.
- **Fila limitada lastRates no endpoint /convert:** Trata-se de uma fila em memória com tamanho restrito (tamanho: 10), o que gera um histórico reduzido e potencialmente insuficiente para refletir variações reais das taxas de conversão. Além disso, o valor padrão fixo (5.0) pode não representar adequadamente cenários reais.
- **Processamento de requisições pendentes no endpoint /bonus:** Não há mecanismo para armazenamento das requisições que permanecem pendentes na fila, o que pode resultar em perda definitiva desses dados caso a instância seja reiniciada ou apresente falha.
- **Ausência de backoff exponencial no endpoint /bonus:** O sistema tenta sempre enviar uma requisição a cada 30 segundos, causando sobrecarga em um serviço caído. É um caso específico para o nosso projeto, haja visto que o endpoint /bonus possui um erro do tipo: crash.

### 3.4 Resumo Final

A implementação atual adota medidas básicas e práticas úteis (retries, cache local, fallback de taxa, timeout curto) e permite validar resiliência com failure-simulator. Para um ambiente de produção, recomenda-se complementar com circuit breaker, backoff exponencial, cache distribuído e mensageria, adicionando mais robustez e consistência.