Capítulo 21 – Fechamento e Ajustes do Back-end Especialista (TP 2)

Aléssio Miranda Jr.

Outubro - 2026/2

Capítulo em Revisão

Este conteúdo está sendo revisado. Algumas informações podem estar preliminares, conter erros de formatação ou sofrer alterações na versão final.

O que você verá neste capítulo

Chegamos ao fechamento do Arco 3. O objetivo do **Trabalho Prático 2 (TP2)** não é apenas montar um RAG — isso já fizemos. O objetivo é montar um RAG que **não quebre em produção**. Neste capítulo, formalizaremos os três pilares da **Engenharia de Resiliência** para sistemas que dependem de APIs externas: *Retry*, *Circuit Breaker* e *Rate Limiting*. Esses padrões separam o código de um estagiário do código de um engenheiro sênior.

1 Introdução: Quando a Nuvem Falha

O paradigma de usar inteligência artificial de terceiros (*OpenAI*, *Google*, *Anthropic*) nos introduz a uma nova variável de caos: e se o servidor deles cair? E se a sua cota de requisições estourar (o temido erro HTTP 429 — *Too Many Requests*)? E se o servidor deles demorar mais de 30 segundos para responder (erro de *Timeout*)?

Uma arquitetura de *back-end* júnior repassa o erro diretamente para a cara do usuário. Uma arquitetura de *back-end* sênior trata o erro, loga a falha e aciona um plano B (*Fallback*).

△ Importante: title=Falhas Reais de Provedores de IA

- **Março 2024:** A API da OpenAI ficou instável por 8 horas, retornando erros 500 intermitentes. Sistemas sem Retry falharam silenciosamente para milhares de usuários.
- **Junho 2024:** O Google Gemini implementou limites mais rígidos de Rate Limiting (15 req/min no tier gratuito). Aplicações que não gerenciavam filas colapsaram.
- **Cenário comum:** O provedor atualiza o modelo, e a resposta muda de formato. Sistemas sem validação de output quebram sem aviso.

2 Engenharia de Resiliência

2.1 Padrão 1: Retry (Tentar Novamente)

Se a API do Gemini retornar um erro 500 (*Internal Server Error*) por uma falha momentânea de rede, seu código não deve falhar instantaneamente. Ele deve aguardar e tentar novamente de forma transparente.

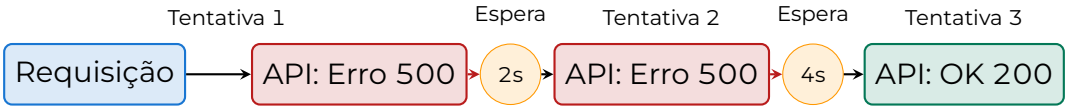

Figura 1: Padrão Retry com **Exponential Backoff**: o tempo de espera dobra a cada falha (2s, 4s, 8s...), evitando sobrecarregar o servidor durante uma degradação parcial.

▶ Prática: title=Retry com Spring Retry

```java
@Service
public class ChatService {

    @Retryable(
        retryFor = {HttpServerErrorException.class},
        maxAttempts = 3,
        backoff = @Backoff(delay = 2000, multiplier = 2)
    )
    public String perguntar(String pergunta) {
        return chatClient.prompt()
            .user(pergunta).call().content();
    }

    @Recover
    public String fallback(Exception e, String pergunta) {
        return "Sistema temporariamente indisponivel. "
             + "Tente novamente em alguns minutos.";
    }
}
```

O `@Retryable` tenta até 3 vezes com backoff exponencial (2s, 4s). Se todas falharem, o `@Recover` retorna uma mensagem amigável.

2.2 Padrão 2: Circuit Breaker (Disjuntor)

Se a OpenAI caiu globalmente, não adianta fazer 1.000 requisições seguidas que vão falhar e sobrecarregar o seu servidor. O "Disjuntor" desarma, bloqueando temporariamente novas chamadas.

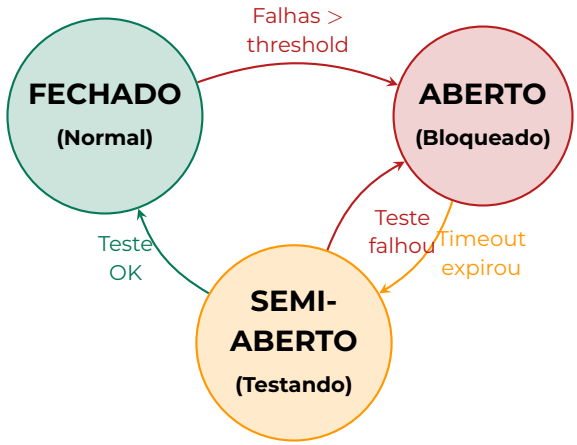

Figura 2: Máquina de estados do Circuit Breaker. O disjuntor "desarma" (ABERTO) após um limiar de falhas, bloqueia todas as requisições por um tempo, e depois testa cautelosamente (SEMI-ABERTO) antes de reabrir.

• Teoria: title=A Analogia Elétrica

O Circuit Breaker funciona exatamente como um disjuntor elétrico doméstico. Quando a corrente (número de falhas) excede o limite seguro, o disjuntor "desarma" para proteger o circuito (seu servidor). Após um período de resfriamento, ele testa com uma pequena carga antes de restabelecer completamente a conexão.

2.3 Padrão 3: Rate Limiting

O seu provedor de IA pode limitar você a 15 requisições por minuto. Seu backend deve gerenciar uma fila ou rejeitar controladamente o excedente antes mesmo de repassá-lo para a nuvem.

Tabela 1: Limites de Rate Limiting dos principais provedores (tier gratuito/básico).

Provedor	Req/minuto	Tokens/minuto
OpenAI (GPT-4o, Tier 1)	500	30.000
Google Gemini (Free)	15	32.000
Google Gemini (Pay-as-you-go)	1.000	4.000.000
Anthropic (Claude 3.5)	50	40.000
Ollama (Local)	∞	Limitado pela GPU

3 Trabalho Prático 2: O Domínio de Dados Restrito

No TP2, as duplas deverão escolher um nicho de dados específico e construir um sistema RAG corporativo completo.

Tabela 2: Exemplos de domínios e fontes de dados para o TP2.

Domínio	Fonte de Dados	Complexidade
Leis de Trânsito	CTB (PDF oficial)	Média
Regras de RPG	Manual D&D 5e (PDF)	Alta (vocabulário específico)
Manuais de Peças	Catálogos industriais (PDF)	Alta (tabelas + imagens)
Receitas Culinárias	Blog compilado (HTML)	Baixa
Normas ABNT	Coletânea de normas (PDF)	Média

★ Checkpoint do Projeto: Critérios de Avaliação do TP2

1. **Qualidade do RAG (40%):** Respostas fundamentadas nos documentos, sem alucinações. Testado com 10 perguntas pré-definidas pelo avaliador.
2. **Resiliência (30%):** O sistema deve sobreviver a:
   - Erro 500 simulado (Retry funcional)
   - Burst de 50 requisições em 10 segundos (Rate Limiting)
   - Desligamento do provedor por 30 segundos (Circuit Breaker + Fallback)
3. **Arquitetura e Código (20%):** Injeção de dependências, separação de camadas, variáveis de ambiente para chaves.
4. **Interface de Usuário (10%):** Front-end funcional consumindo a API via HTTP.

4 A Arquitetura Completa do TP2

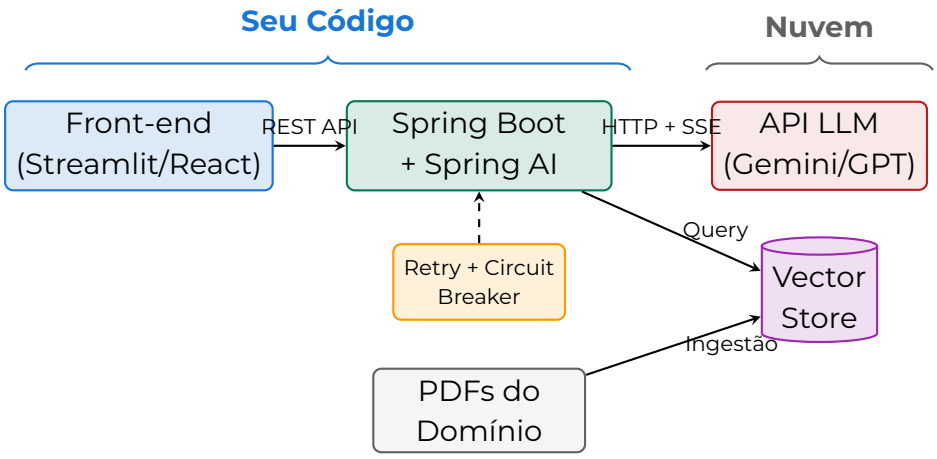

Figura 3: Arquitetura completa do sistema TP2: Front-end → Spring Boot (com camada de resiliência) → API LLM + Vector Store.

5 Conclusão do Arco 3

Com a entrega do TP2, vocês consolidam o Arco de Integração Corporativa. A Inteligência Artificial Generativa deixou de ser um chat mágico para se tornar um **componente arquitetural** do sistema de vocês, com chaves de acesso escondidas, CORS habilitado, injeção de dependências orquestrando o fluxo vetorial e tratamento de exceções garantindo a paz do servidor.

✓ O que Vocês Dominam Agora

✓ Chamar APIs de LLMs com segurança (chaves em variáveis de ambiente)

✓ Engenharia de Prompts (Zero/Few-Shot, CoT, JSON Mode)

✓ RAG completo (Ingestão → Chunking → Embedding → Retrieval → Generation)

✓ Integração corporativa com Spring AI (ChatClient, Advisors, VectorStore)

✓ Interface de usuário com streaming (SSE)

✓ Resiliência (Retry, Circuit Breaker, Rate Limiting)

Estão prontos para o nível avançado: **Agentes Autônomos**.

✔ Takeaways

- **Retry com Exponential Backoff** trata falhas momentâneas da API: o tempo de espera dobra a cada tentativa (2s, 4s, 8s), evitando sobrecarregar o servidor degradado.
- O **Circuit Breaker** funciona como um disjuntor elétrico: após um limiar de falhas, bloqueia temporariamente todas as requisições (estado ABERTO), testa cautelosamente (SEMI-ABERTO) e só reabre quando o serviço se recupera.
- **Rate Limiting** protege contra o estouro de cota: o back-end deve gerenciar filas ou rejeitar o excedente antes de repassar ao provedor de IA na nuvem.
- A anotação `@Retryable` do Spring combina com `@Recover` para implementar fallbacks amigáveis quando todas as tentativas falham.
- A resiliência separa o código de um estagiário (“erro 500 na cara do usuário”) do código de um engenheiro sênior (retry + fallback + logging).
- O TP2 avalia: qualidade do RAG (40%), resiliência (30%), arquitetura (20%) e interface (10%).

✎ Questões: Autoavaliação

1 **[Direta]** Descreva os três estados do Circuit Breaker (Fechado, Aberto, Semi-Aberto) e explique as condições de transição entre eles.

2 **[Direta]** No código com `@Retryable`, o que os parâmetros `maxAttempts=3` e `backoff = @Backoff(delay=2000, multiplier=2)` significam em termos de tempo total de espera?

3 **[Direta]** A tabela mostra que o Google Gemini Free permite 15 req/min. Se sua aplicação tem 5 usuários simultâneos enviando 1 pergunta a cada 2 minutos, o rate limit será atingido? Justifique com cálculos.

4 **[Reflexiva]** Compare a abordagem de resiliência do Spring (Retry + Circuit Breaker em Java) com a abordagem de "try/except" simples em Python. Quais falhas sutis o Python puro não consegue tratar?

5 **[Reflexiva]** Um sistema de triagem médica usa RAG e a API do Gemini cai por 2 horas. O fallback retorna "Sistema indisponível". Discuta as implicações éticas de um sistema de saúde crítico depender de APIs de terceiros e proponha arquiteturas alternativas.

Lab. 21: Resiliência na API com Retry e Fallback

Capítulo em Revisão

Este conteúdo está sendo revisado. Algumas informações podem estar preliminares, conter erros de formatação ou sofrer alterações na versão final.

⋆ Objetivo do Laboratório

APIs de LLM na nuvem falham — é uma certeza, não uma possibilidade. Neste laboratório, você transformará o endpoint RAG do Lab 19 em um serviço **resiliente**: adicionará `@Retryable` com Exponential Backoff para falhas transitórias, um método `@Recover` como fallback amigável, e simulará cenários de erro para comprovar que o sistema **nunca** retorna um `500 Internal Server Error` ao usuário final.

6 Parte 1: Adicionando a Dependência de Retry (5 min)

No projeto Spring Boot do Lab 19, adicione a dependência de retry ao `pom.xml`:

```
<dependency>
    <groupId>org.springframework.retry</groupId>
    <artifactId>spring-retry</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aspects</artifactId>
</dependency>
```

Na classe principal (`@SpringBootApplication`), adicione a anotação de habilitação:

```
@SpringBootApplication
@EnableRetry  // Habilita o mecanismo de retry do Spring
public class SpringAiDemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(SpringAiDemoApplication.class, args);
    }
}
```

7 Parte 2: Criando o Service Resiliente (15 min)

Crie a classe `ChatService.java` para separar a lógica de negócio do Controller (arquitetura limpa):

```java
package br.edu.instituicao.springaidemo;

import org.springframework.ai.chat.client.ChatClient;
import org.springframework.ai.chat.client.advisor.QuestionAnswerAdvisor;
import org.springframework.ai.vectorstore.VectorStore;
import org.springframework.retry.annotation.Backoff;
import org.springframework.retry.annotation.Recover;
import org.springframework.retry.annotation.Retryable;
import org.springframework.stereotype.Service;

@Service
public class ChatService {

    private final ChatClient chatClient;

    public ChatService(ChatClient.Builder builder, VectorStore vectorStore) {
        this.chatClient = builder
                .defaultAdvisors(new QuestionAnswerAdvisor(vectorStore))
                .build();
    }

    @Retryable(
        maxAttempts = 3,
        backoff = @Backoff(delay = 2000, multiplier = 2)
        // Tentativa 1: imediata
        // Tentativa 2: espera 2s
        // Tentativa 3: espera 4s
    )
    public String perguntar(String mensagem) {
        return chatClient.prompt()
                .user(mensagem)
                .call()
                .content();
    }

    @Recover
    public String fallback(Exception e, String mensagem) {
        return "Desculpe, nosso sistema de IA esta temporariamente "
             + "indisponivel. Tente novamente em alguns instantes. "
             + "(Erro: " + e.getClass().getSimpleName() + ")";
    }
}
```

8 Parte 3: Atualizando o Controller (5 min)

Simplifique o `ChatController.java` para delegar ao Service:

```java
@RestController
@CrossOrigin(origins = "*")
public class ChatController {

    private final ChatService chatService;

    public ChatController(ChatService chatService) {
        this.chatService = chatService;
    }

    @GetMapping("/api/rag")
    public String perguntaRag(@RequestParam String msg) {
        return chatService.perguntar(msg);
    }
}
```

9 Parte 4: Testando a Resiliência (10 min)

A. Teste Normal

1. Rode a aplicação e acesse: `http://localhost:8080/api/rag?msg=Qual o vale-refeição?`
2. Confirme que a resposta é correta e fundamentada nos documentos do VectorStore.

B. Simulando Falha

1. Altere temporariamente a chave da API para um valor inválido: `spring.ai.openai.api-key=INVALIDA`.
2. Reinicie o servidor e faça a mesma pergunta.
3. Observe nos logs do Spring: três tentativas com delays crescentes (2s, 4s), seguidas da mensagem do `@Recover`.
4. O usuário **nunca** vê um `500 Internal Server Error` — ele recebe a mensagem amigável do fallback.

△ Importante: Erros Comuns

- **Esqueceu** `@EnableRetry`: O `@Retryable` é silenciosamente ignorado sem essa anotação na classe principal.
- **O método** `@Recover` **tem assinatura errada**: O primeiro parâmetro deve ser a **Exception**, e os demais devem ser idênticos aos do método anotado com `@Retryable`.
- **Retry e Recover no mesmo componente**: Ambos devem estar na mesma classe (`ChatService`), ou o proxy do Spring não intercepta corretamente.

10 Parte 5: Commit e Push (5 min)

1. Restaure a chave de API correta no `application.properties`.
2. Adicione os arquivos modificados: `git add src/`.
3. Crie o commit: `git commit -m "Lab 21: Retry + Fallback resiliente no RAG"`.
4. Envie para o GitLab: `git push origin main`.

Conclusão: O que Você Construiu Hoje

✔ Takeaways

- Você separou a lógica de negócio em um `@Service` — padrão MVC que o mercado corporativo exige.
- Implementou `@Retryable` com **Exponential Backoff** (2s, 4s) — o sistema agora tolera falhas momentâneas da API sem desistir na primeira tentativa.
- Criou um `@Recover` (fallback) que garante que o usuário **jamais** verá uma exceção crua — ele recebe uma mensagem amigável.
- Testou manualmente a resiliência simulando uma chave inválida e observando o comportamento nos logs.
- A separação Controller → Service é a mesma arquitetura usada em bancos, fintechs e e-commerces que operam 24/7.

◇ **Informação**

Gancho para o TP2: Este padrão de resiliência é **obrigatório** no Trabalho Prático 2. Na próxima aula, vocês terão tempo de desenvolvimento dedicado para integrar todos os componentes (RAG + Front-end + Resiliência) em um produto funcional completo.