Capítulo 20 – O “Fator Uau!” (Consumindo a API)

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

Uma API REST invisível não impressiona ninguém. Neste capítulo, construiremos a camada de **interface de usuário** que consome o back-end RAG, transformando chamadas HTTP em uma experiência visual interativa. Exploraremos duas abordagens: um front-end leve com `Streamlit` (Python) e uma interface web com `React/HTML + JavaScript`. Abordaremos o conceito de *streaming* de respostas (Server-Sent Events), CORS, e os princípios de UX para interfaces de chat com IA.

1 Da API ao Produto Visual

Nas últimas aulas, construímos um back-end poderoso: um sistema RAG com Spring AI que busca documentos no Vector Store, enriquece o prompt e gera respostas fundamentadas. Mas para o usuário final (cliente, gestor, paciente), o back-end é invisível. O que ele vê é a **interface**.

A experiência do ChatGPT provou que a interface “de chat” é intuitiva e universal. Vamos replicar esse padrão para o nosso sistema.

2 Abordagem 1: Streamlit (Prototipagem Rápida)

O **Streamlit** é um framework Python que cria interfaces web interativas com poucas linhas de código:

▶ Prática: title=Chat UI com Streamlit

```python
import streamlit as st
import requests

st.title("Assistente IA - RAG Corporativo")

# Historico de mensagens
if "msgs" not in st.session_state:
    st.session_state.msgs = []

# Exibir historico
for msg in st.session_state.msgs:
    with st.chat_message(msg["role"]):
        st.write(msg["content"])

# Input do usuario
if pergunta := st.chat_input("Faca sua pergunta..."):
    st.session_state.msgs.append(
        {"role": "user", "content": pergunta}
    )

    # Chamar a API Spring AI
    resp = requests.post(
        "http://localhost:8080/perguntar",
        data=pergunta
    )

    st.session_state.msgs.append(
        {"role": "assistant", "content": resp.text}
    )
    st.rerun()
```

Execute com `streamlit run app.py` e uma interface de chat profissional aparece no navegador.

3 Abordagem 2: Interface Web (HTML + JavaScript)

Para integração com sistemas web existentes, uma página HTML simples que consome a API via `fetch`:

```html
<!-- index.html -->
<div id="chat-container">
    <div id="messages"></div>
    <input type="text" id="input"
           placeholder="Faca sua pergunta...">
    <button onclick="enviar()">Enviar</button>
</div>
```

```
8
9  <script>
10 async function enviar() {
11     const input = document.getElementById('input');
12     const pergunta = input.value;
13
14     const resp = await fetch('/perguntar', {
15         method: 'POST',
16         body: pergunta
17     });
18
19     const resposta = await resp.text();
20     adicionarMensagem('assistant', resposta);
21     input.value = '';
22 }
23 </script>
```

4 Streaming de Respostas (SSE)

Uma resposta do LLM pode levar 5 a 15 segundos para ser gerada completamente. Ficar olhando uma tela em branco por 15 segundos é uma experiência terrível. A solução é o **Streaming**: enviar a resposta *token por token* conforme ela é gerada, exibindo o texto "digitando" em tempo real.

O padrão utilizado é o **Server-Sent Events (SSE)**, onde o servidor envia fragmentos de dados em uma conexão HTTP persistente:

```
1 // Spring AI com Streaming
2 @GetMapping(value = "/stream",
3             produces = MediaType.TEXT_EVENT_STREAM_VALUE)
4 public Flux<String> stream(@RequestParam String pergunta) {
5     return chatClient.prompt()
6         .user(pergunta)
7         .stream()
8         .content();
9 }
```

• Teoria: title=UX de Chat com IA — Boas Práticas

1. **Indicador de "digitando":** Exibir animação enquanto o LLM processa.
2. **Citação das fontes:** Mostrar quais documentos do Vector Store foram usados na resposta.
3. **Feedback do usuário:** Botões de "like/dislike" em cada resposta para avaliação humana.
4. **Limitar expectativas:** Disclaimers como "Esta é uma resposta gerada por IA e pode conter imprecisões".

5 CORS: O Bloqueio Silencioso

Quando o front-end (porta 3000) tenta acessar o back-end (porta 8080), o navegador bloqueia a requisição por segurança (*Cross-Origin Resource Sharing*). No Spring Boot, habilitamos o CORS com:

```
@Configuration
public class CorsConfig implements WebMvcConfigurer {
    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/**")
            .allowedOrigins("http://localhost:3000")
            .allowedMethods("GET", "POST");
    }
}
```

△ Importante: title=Nunca Use allowedOrigins("*") em Produção

Permitir qualquer origem (*) abre a API para ser consumida por qualquer site na internet, incluindo sites maliciosos. Em produção, liste explicitamente os domínios autorizados.

6 Conclusão

Com a camada visual finalizada, o sistema RAG deixou de ser uma API invisível para se tornar um **produto utilizável**. As próximas duas aulas (21–22) serão dedicadas ao desenvolvimento do **Trabalho Prático 2 (TP2)**, onde as duplas construirão um sistema RAG completo (back-end Spring AI + front-end) sobre um domínio de dados real à sua escolha.

✔ Takeaways

- A **interface de usuário** é o que transforma uma API invisível em um produto — usuários não compram endpoints, compram experiências visuais.
- **Streamlit** é a rota mais rápida para prototipagem de interfaces de IA (poucas linhas de Python), enquanto **React/HTML+JS** oferece controle total para produção.
- O **Streaming via SSE** (Server-Sent Events) elimina a percepção de demora ao exibir a resposta token por token, imitando a experiência do ChatGPT.
- **CORS** é o bloqueio mais comum em integrações front-end ↔ back-end. É resolvido exclusivamente no lado do servidor, nunca no HTML.
- Boas práticas de UX para IA incluem: indicadores de "digitando", citação de fontes, botões de feedback (like/dislike) e disclaimers de limitação.
- Nunca use `allowedOrigins("*")` em produção — isso abre a API para consumo por qualquer site na internet.

✎ Questões: Autoavaliação

1 **[Direta]** Explique a diferença técnica entre uma resposta HTTP convencional e o Streaming via SSE. Quais classes do Spring AI habilitam o streaming?

2 **[Direta]** O que é CORS e por que o navegador bloqueia requisições entre origens diferentes? Como resolver o bloqueio no Spring Boot?

3 **[Direta]** Compare as vantagens e desvantagens do Streamlit versus React para construir uma interface de chat com IA. Quando cada escolha é mais adequada?

4 **[Reflexiva]** Um sistema RAG demora 12 segundos para responder. Sem streaming, o usuário vê uma tela em branco. Proponha três estratégias de UX para manter o engajamento durante a espera.

5 **[Reflexiva]** Discuta a importância de exibir as fontes (documentos do Vector Store) usadas na resposta do RAG. Que impacto isso tem na confiança do usuário e na auditabilidade do sistema?

Lab. 20: Conectando o Chat (UI)

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

Habilitar o CORS na nossa API Spring Boot e construir um HTML muito simples com JavaScript para consumir a rota RAG criada no laboratório passado, proporcionando a experiência completa de uso (*Fator Uau!*).

7 Atividade Prática

1. Liberando o CORS no Java

Abra o projeto do Laboratório 19 na sua IDE. Vá até a classe `ChatController.java` e adicione a anotação `@CrossOrigin` acima da classe para permitir que chamadas do navegador sejam aceitas.

```
import org.springframework.web.bind.annotation.CrossOrigin;
// ... (outros imports)

@RestController
@CrossOrigin(origins = "*") // ATENÇÃO: Libera de qualquer origem (Apenas para fins di
public class ChatController {
    // ... código restante
}
```

Re-execute o projeto Spring Boot.

2. O Código Front-end

Crie uma nova pasta em qualquer lugar do seu computador (fora do projeto Java) e crie um arquivo chamado `index.html`:

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
    <meta charset="UTF-8">
```

```
    <title>Chat com IA da Empresa</title>
    <style>
        body { font-family: sans-serif; padding: 20px; max-width: 600px; margin: auto
        #chat-box { border: 1px solid #ccc; padding: 10px; min-height: 200px; margin-b
        .user { color: blue; font-weight: bold; }
        .ai { color: green; margin-bottom: 15px; }
    </style>
</head>
<body>

    <h2>Chatbot RAG Corporativo</h2>
    <div id="chat-box"></div>
    <input type="text" id="pergunta" placeholder="Digite sua pergunta..." style="width
    <button onclick="enviarPergunta()">Enviar</button>

    <script>
        async function enviarPergunta() {
            const input = document.getElementById("pergunta");
            const chatBox = document.getElementById("chat-box");
            const pergunta = input.value;

            if(!pergunta) return;

            // Imprime a pergunta do usuário na tela
            chatBox.innerHTML += `<div class="user">Você: ${pergunta}</div>`;
            chatBox.innerHTML += `<div class="ai" id="loading">Pensando...</div>`;
            input.value = ""; // Limpa a caixa

            try {
                // Requisição FETCH para o nosso Back-end Spring Boot
                const url = `http://localhost:8080/api/rag?msg=${encodeURIComponent(pe
                const resposta = await fetch(url);
                const textoGerado = await resposta.text();

                // Remove o loading e insere a resposta final
                document.getElementById("loading").remove();
                chatBox.innerHTML += `<div class="ai">IA: ${textoGerado}</div>`;
            } catch (erro) {
                document.getElementById("loading").remove();
                chatBox.innerHTML += `<div class="ai" style="color:red;">Erro ao conec
            }
        }
    </script>
```

```
</body>
</html>
```

3. Execução e Entrega

1. Com o servidor Spring rodando, abra o `index.html` diretamente no seu navegador (com um duplo-clique).
2. Faça a pergunta que configuramos no *VectorStore* (ex: sobre o vale-refeição) e aguarde a resposta visual.
3. Comemore a integração *Full-Stack* finalizada!
4. Envie o código modificado do Java (com a anotação do CORS) e o arquivo `index.html` para o GitLab.

8 Critério de Avaliação

Entrega consolidada do HTML operando o `fetch` e o `ChatController` Java aceitando a requisição HTTP.

✔ Takeaways

- Você realizou a **integração Full-Stack completa**: front-end (HTML+JS) ↔ back-end (Spring AI) ↔ LLM (nuvem).
- O `@CrossOrigin` resolveu o bloqueio de CORS — o navegador agora aceita requisições para o back-end Java.
- O `fetch()` assíncrono com `async/await` consome a API REST sem recarregar a página — o padrão SPA moderno.
- O indicador “Pensando...” melhora a UX ao dar feedback imediato ao usuário enquanto o LLM processa a resposta.

◇ Informação

Gancho para a Próxima Aula: O sistema funciona, mas é frágil: se a API da OpenAI/Gemini cair, o usuário verá um erro cru no navegador. Na próxima aula, adicionaremos **Resiliência** com `@Retryable`, Exponential Backoff e fallback amigável — o que separa um protótipo de um produto de verdade.