Inteligência Artificial 2

Aula 20: O "Fator Uau!" (Consumindo a API)

Prof. Aléssio Miranda Júnior
alessio@cefetmg.br

CEFET-MG - Campus Timoteo
Dep. Engenharia de Computação

Inteligência Artificial 2

Sumário

Onde Paramos?

Na aula anterior, construímos o **RAG corporativo completo em Spring AI**:

- Pipeline ETL: `DocumentReader` → `TokenTextSplitter` → `VectorStore`.
- `QuestionAnswerAdvisor` interceptando e aumentando prompts.
- Endpoint `/api/rag` respondendo com dados fundamentados.

A Pergunta de Hoje

O RAG funciona no Postman. Mas **usuários reais** não usam Postman — como dar um **rosto** ao sistema?

Objetivos da Sessão

Ao final desta aula, você será capaz de:

1. **Configurar** CORS no Spring Boot para aceitar requisições do navegador.
2. **Construir** uma interface HTML+JavaScript que consome a API REST.
3. **Implementar** `fetch()` assíncrono com indicador de carregamento.
4. **Explicar** a arquitetura Full-Stack: Front $\leftrightarrow$ Back $\leftrightarrow$ LLM.
5. **Integrar** todas as peças construídas desde a Aula 15.

A Ilusão do Terminal

- Até este ponto, operamos estritamente no ecossistema do **Back-end** (Servidor).
- Vimos a mágica matemática dos vetores e das probabilidades se concretizar no *Console* (Terminal) e por meio de ferramentas de depuração como o *Postman* ou *cURL*.
- Porém, para a imensa maioria dos usuários finais (clientes corporativos, leigos ou *stakeholders*), a infraestrutura *backend* é invisível.
- O valor de negócio (o "Fator Uau") só se tangibiliza quando o sistema ganha um **Front-end** (uma Interface Gráfica interativa).

A Arquitetura Full-Stack com IA

Aplicações de Inteligência Artificial modernas seguem invariavelmente uma topologia de Microsserviços ou API-First.

- **Apresentação (Front-end):** SPA (*Single Page Application*) rodando no navegador do cliente (React, Vue, ou JS "Vanilla"). Controla UX, cores, *spinners* de carregamento e acessibilidade.
- **Orquestração (Back-end):** O nosso Spring Boot. Ele recebe chamadas HTTP do *frontend*, injeta o conhecimento do *VectorStore* (ChromaDB) via *Advisors* e chama a Nuvem (OpenAI/Google).
- O modelo em nuvem devolve ao Spring, que sanitiza o dado e envia como resposta HTTP de volta para o navegador desenhar.

Protocolo HTTP: O Mensageiro Universal

Para que a tela web converse com o *backend* em Java, a tecnologia subjacente é o Protocolo HTTP (*Hypertext Transfer Protocol*).

- O navegador envia um **Request** (`GET` ou `POST`) contendo a *Query* digitada no *input* HTML.
- O servidor processa e retorna um **Response** (*Status Code* 200 OK), frequentemente acoplado a um *payload* textual (JSON).
- **Por que JSON?** O *JavaScript Object Notation* é o "Esperanto" da web. O Java (fortemente tipado) escreve o JSON, e o *JavaScript* (fracamente tipado) lê o JSON nativamente como um objeto acessível.

A Assincronicidade do JavaScript

No *Frontend*, chamadas para Inteligências Artificiais demoram (de dezenas de milissegundos até dezenas de segundos).

- O JavaScript é *Single-Threaded* (possui apenas um fio principal de execução).
- Se a chamada HTTP fosse "blocante", a tela inteira do usuário "congelaria" até o LLM terminar de pensar. Botões não clicariam, o *scroll* travaria.
- A solução arquitetural é a programação **Assíncrona**.

Promises e o Event Loop

O *Event Loop* (Laço de Eventos) delega requisições HTTP para as Web APIs do navegador e continua livre para desenhar a interface.

- Uma **Promise** (Promessa) é um objeto que representa a eventual conclusão de uma operação assíncrona.
- A sintaxe moderna `async / await` permite escrever código assíncrono que "parece" síncrono e fácil de ler, sem causar travamentos na UI (User Interface).

Consumindo a API: A Função `fetch()`

A API nativa dos navegadores modernos para disparar requisições HTTP é a função `fetch`.

```
// Exemplo em JavaScript (Front-end)
async function buscarRespostaIA(perguntaUsuario) {
    const url = "http://localhost:8080/api/chat?q=" +
        encodeURIComponent(perguntaUsuario);

    // A thread pausa AQUI apenas logicamente, mas a tela continua
        viva
    const response = await fetch(url);

    // Transforma a resposta crua em texto ou JSON
    const data = await response.text();
    return data;
}
```

Manipulação do DOM (Document Object Model)

- O *Document Object Model* é a árvore hierárquica de elementos HTML em memória que o navegador renderiza.
- Quando a *Promise* da IA se resolve, usamos comandos do JS (`document.getElementById()`) para capturar `<div>` ou `<p>` e reescrever sua propriedade `innerHTML`.
- É este ato de mutação no DOM que o usuário percebe como "A IA respondeu à minha tela".

Além do Vanilla: O Virtual DOM (React)

- Embora a manipulação direta do DOM (Vanilla JS) funcione bem para provas de conceito, em aplicações massivas de IA ela se torna insustentável.
- *Frameworks* modernos como **React** utilizam o conceito de *Virtual DOM*. Eles re-renderizam os componentes automaticamente baseado em uma variável de estado (ex: `useState`).
- Quando a resposta da API muda o estado da "mensagem atual", o React calcula o *diff* (diferença) e atualiza apenas aquele pedaço específico da tela com performance absoluta, muito mais eficiente que sobrescrever grandes blocos de `innerHTML`.

Feedback Visual (UX de Latência)

Em IAs generativas, a latência de inferência é o calcanhar de Aquiles.

- Um *Request* RAG corporativo faz: 1. *Embedding* da Pergunta; 2. Busca Vetorial de milhões de registros; 3. Chamada REST para a OpenAI; 4. Raciocínio autorregressivo do LLM gerando 1.000 tokens; 5. Serialização de volta.
- Isso pode durar de 5 a 20 segundos.
- **Regra de Prata de UX:** Antes de disparar o `fetch`, manipule o DOM injetando um indicador visual (um *Loading Spinner* ou um texto "A IA está pensando..."). Sem isso, o usuário inferirá que a página está quebrada e apertará "Enviar" dezenas de vezes, derrubando a API (DDoS acidental).

O Pesadelo dos Juniores: Erro CORS

Assim que o código JavaScript do Front-end dispara o `fetch()` para `http://localhost:8080/api/chat`, a requisição costuma falhar catastróficamente com uma mensagem em vermelho no Console do Navegador:

"Access to fetch at 'localhost:8080' from origin '127.0.0.1:5500' has been blocked by CORS policy."

Este não é um defeito do código. É o mecanismo de defesa padrão do navegador Web em pleno funcionamento.

O Conceito de "Same-Origin Policy" (SOP)

A Política de Mesma Origem dita que scripts rodando em um "Site A" (`dominio-A.com`) não podem invocar APIs livremente de um "Site B" (`api-do-banco.com`).

- Se o SOP não existisse, um site malicioso poderia disparar um `fetch` oculto para o banco onde o usuário está logado e roubar dados sem ele saber (*Cross-Site Request Forgery*).
- A regra relaxa apenas se o *Backend* explicitamente anunciar "Eu confio em requisições vindas do Site A". O nome dessa declaração de relaxamento é **CORS (Cross-Origin Resource Sharing)**.

Requisições Preflight (O Método OPTIONS)

- Quando a UI tenta postar dados customizados via `fetch`, o navegador não manda a requisição inteira direto.
- Ele realiza uma chamada furtiva chamada **Preflight Request** usando o método HTTP `OPTIONS`.
- O navegador pergunta silenciosamente ao servidor Spring: *"O domínio localhost:5500 tem permissão para acessar a rota X usando POST?"*
- Se o Spring Boot não estiver configurado para responder positivamente, o navegador bloqueia e "rasga" a chamada antes do servidor processar o prompt LLM.

CORS: Desenvolvimento vs. Produção

A configuração e os riscos do CORS mudam radicalmente dependendo do ambiente.

- **Ambiente de Desenvolvimento:** É aceitável usar `@CrossOrigin(origins = "*")` para não travar o time de *Frontend* que roda serviços locais variados.
- **Ambiente de Produção:** A aplicação Spring e o Frontend (React/Vue) frequentemente são servidos sob o **mesmo domínio principal** (ex: `app.empresa.com` e `api.empresa.com`), o que flexibiliza o CORS se configurado adequadamente via *API Gateway* ou instâncias *Nginx*.
- Máquinas e servidores não executam *Preflight*. O erro de CORS **só existe no Navegador**. Se você rodar o mesmo *Endpoint* RAG no Postman, ele sempre funcionará perfeitamente!

Resolvendo CORS no Spring Boot

A configuração de origens confiáveis é feita trivialmente no *framework* Java através da anotação `@CrossOrigin`.

```
RestControllerRequestMapping("/api/chat")
// Libera o acesso para qualquer porta localhost
CrossOrigin(origins = "*")public class ChatController // ...  codigo do ChatClient
```

Em Produção, a origem `"*"` (*Wildcard*) é evitada, sendo substituída pelas URLs exatas em que o Frontend da empresa está hospedado (ex: `"https://meu-app-front.com.br"`).

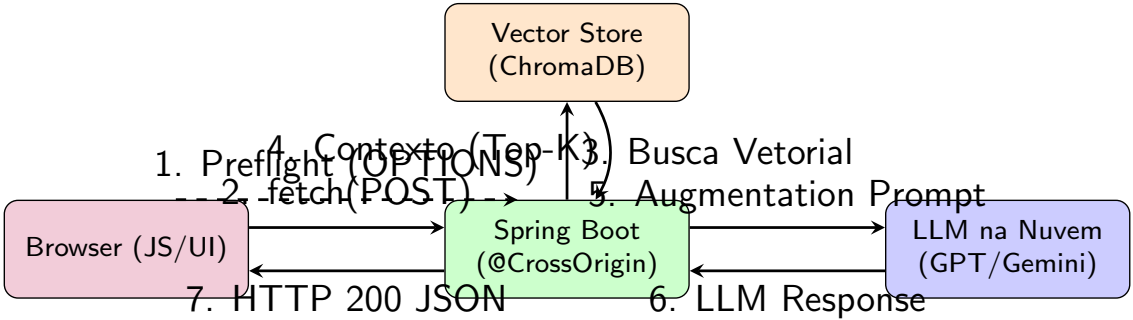
Streaming vs Blocante

- O fluxo REST tradicional (`fetch` $\rightarrow$ Espera Completa $\rightarrow$ Pinta na tela) provoca degradação perceptível se a resposta for longa.
- **A Ilusão do ChatGPT:** Por que o ChatGPT "digita" a resposta letra por letra na tela? Não é apenas um "charme" visual, é a exibição em **Streaming**.
- Uma rede neural geradora (*Decoder*) gera os *tokens* sequencialmente. O processamento da palavra número 2 não espera a 500.

Server-Sent Events (SSE) para IAs

- APIs de IA modernas suportam fluxos contínuos. No Java (Spring AI), convertemos o `call()` para `stream()`.
- A comunicação passa a usar **Server-Sent Events (SSE)** (Fluxo *Flux¡String¿*).
- A conexão HTTP fica mantida "aberta". Assim que a IA "cospe" a palavra *"Olá,"*, o Spring injeta ela na conexão; o *JavaScript* no Front-end captura esse pedaço imediatamente e o anexa ao HTML, enquanto a próxima palavra ainda está sendo decodificada.
- O TTFT (*Time to First Token* - Tempo até o primeiro caractere) despenca de 10 segundos para menos de 500 milissegundos.

A Anatomia Completa de Requisição



Conclusão: O Engenheiro de IA Completo (End-to-End)

- A Inteligência Artificial em escopo industrial requer fluidez em várias frentes arquitetônicas.
- Um Engenheiro de IA moderno trafega com desenvoltura pelo isolamento de *Back-end* (*RAG, Embeddings, Spring Boot*), contorna obstáculos de redes corporativas (Tratamento de *CORS, Rate Limits*) e providencia garantias de *User Experience* (*UX/UI*) por meio de retornos assíncronos não-blocantes no *Front-end*.
- O ciclo foi concluído, interligando matemática linear vetorial, bancos de dados dinâmicos, rotas seguras e processamento web *Full-Stack*.

Dúvidas?

Email: alessio@cefetmg.br
Aula: Segunda e Quarta-Feira, 7h as 8:40h (Sala 39)
WhatsApp: Dúvidas rápidas

Lembrete

Confira sua Issue no GitLab!

Próxima aula: Métricas RAG (Ragas)

100% da Aula 20 Concluída!