Inteligência Artificial 2

Aula 18: A Virada de Chave (Bem-vindos ao Spring AI)

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 completo em Python puro**:

- ChromaDB para busca vetorial semântica.
- F-string template para injetar contexto no prompt.
- Guardrails: LLM recusa perguntas fora do domínio.

A Pergunta de Hoje

O RAG em Python funciona, mas não escala.
Como migrar para o ecossistema **corporativo Java**?

Objetivos da Sessão

Ao final desta aula, você será capaz de:

1. **Justificar** por que empresas escolhem Java/Spring para IA em produção.
2. **Configurar** um projeto Spring Boot com a dependência Spring AI.
3. **Usar** o `ChatClient` via Injeção de Dependências.
4. **Expor** um endpoint REST que consome um LLM na nuvem.
5. **Comparar** a abordagem Python (script) vs Java (enterprise).

A Hegemonia do Python na IA

- Python é, de forma incontestável, a "língua franca" da pesquisa em Inteligência Artificial.
- Bibliotecas primárias de *Deep Learning* (PyTorch, TensorFlow) e ecossistemas de dados (Pandas, NumPy) orbitam quase exclusivamente em torno do Python devido à sua sintaxe fluida e integração profunda com a linguagem C.
- É a ferramenta perfeita para cientistas de dados validarem *Pipelines*, estruturarem Provas de Conceito (PoCs) e explorarem dados através de *Jupyter Notebooks*.

O Vale da Morte (Da PoC à Produção)

- Quando a pesquisa é concluída e o modelo (ou fluxo RAG) precisa escalar para processar 10.000 requisições por segundo num banco de varejo, a adoção do Python encontra atritos.
- Sistemas legados, microsserviços globais, governança de tipagem rigorosa e alta escalabilidade multithreading são tradicionalmente dominados pela **Java Virtual Machine (JVM)**.
- Reescrever ou integrar módulos Python dentro de um ecossistema nativamente Java gera custos arquiteturais massivos, latência de rede adicional (via APIs internas) e silos de desenvolvimento.

O Ecossistema Enterprise (Java e Spring Boot)

- O mercado *Enterprise* financeiro, de saúde e de grande comércio global baseia-se maciçamente no **Spring Boot** (Java/Kotlin).
- Ele oferece segurança nativa avançada (Spring Security), observabilidade (Micrometer) e injeção de dependências robusta de missão crítica.
- Com o *boom* da IA Generativa, havia uma necessidade urgente de invocar modelos LLMs massivos (OpenAI, Gemini) diretamente de controladores e serviços Java, preservando os padrões de design (GoF) clássicos.

O Cenário Fragmentado

- Inicialmente, para consumir LLMs em Java, os desenvolvedores utilizavam `RestTemplate` ou `WebClient` para fazer chamadas HTTP brutas para os *endpoints* da provedora.
- **O Problema:** O payload JSON da OpenAI é completamente diferente do payload da Anthropic, que é diferente do Llama (via Ollama), que difere do Google Vertex AI.
- Construir classes *Data Transfer Object (DTO)* manuais para mapear *responses* de cada IA envenenava o código de negócios com infraestrutura de terceiros.

Acoplamento Forte a Provedores (Vendor Lock-in)

A Armadilha do Vendor Lock-in

Se você construiu milhares de linhas de código usando o *SDK* proprietário de um provedor (ex: OpenAI) e amanhã ele dobra o preço ou sai do ar, sua aplicação inteira fica refém, pois a migração exige uma reescrita do núcleo do sistema.

No frenético mercado de LLMs, o "modelo Estado da Arte" alterna de empresa a cada 3 meses. Aplicações resilientes precisam ser agnósticas ao provedor de IA utilizado.

Princípio da Inversão de Dependência (SOLID)

Em engenharia de software, o *Dependency Inversion Principle* dita que módulos de alto nível (Lógica de Negócios) não devem depender de módulos de baixo nível (Implementação da OpenAI). Ambos devem depender de **Abstrações** (Interfaces).

- Em vez de `OpenAiService.generate()`, o código deve usar `GenericAiModel.chat()`.
- A implementação concreta deve ser injetada externamente em tempo de compilação ou execução.

A Solução: O Projeto Spring AI

Inspirado nos conceitos concebidos pelo *LangChain* e *LlamaIndex* no mundo Python, a VMWare/Tanzu criou o **Spring AI**.

- Não é um LLM. É um *Framework* arquitetural.
- Propósito: Fornecer abstrações unificadas e padronizadas de IA Generativa para o ecossistema Spring Boot.
- O código escrito com Spring AI funciona nativamente com dezenas de provedores (*Bedrock*, *Azure*, *HuggingFace*, *Mistral*) com zero alteração na lógica de negócio.

Abstração de Interfaces de IA

A genialidade do Spring AI reside na hierarquia de interfaces.

- `ChatModel`: A interface fundamental que os fabricantes implementam.
- `EmbeddingModel`: A interface abstrata de vetorização.
- `VectorStore`: A interface abstrata para banco de dados (*ChromaDB*, *pgvector*, *Milvus*). O desenvolvedor manda `save(document)`, não importa quem fará o SQL nos bastidores.

O Núcleo do Spring AI: `ChatModel`

A interface `ChatModel` aceita um `Prompt` genérico contendo múltiplas `Message` e retorna um `ChatResponse`.

- A injeção dessa dependência (`@Autowired`) é automática (*Auto-configured*) baseando-se no que está declarado no gerenciador de pacotes do projeto (`pom.xml` do Maven).
- Se o pacote `spring-ai-openai` estiver na rota de classe (*classpath*), a interface receberá a concretização `OpenAiChatModel`.

O Construtor Fluente: `ChatClient`

Para facilitar o uso corporativo, adotou-se o padrão *Builder* por meio da classe `ChatClient`, proporcionando código limpo e idiomático (*Fluent API*).

```
// Configurando e chamando o LLM com o padrao Fluent
String resposta = chatClient.prompt()
    .system("Voce e um especialista em design de banco de dados.")
    .user("Diferencie NoSQL de SQL.")
    .call()
    .content(); // Extrai apenas a string limpa resultante
```

Padronização de Prompts e Mensagens

A estrutura de *Roles* que estudamos no capítulo 16 (*System*, *User*, *Assistant*) foi transposta rigidamente para o Java por meio de Herança de Classes:

- Classe `SystemMessage`: Protege o contorno comportamental.
- Classe `UserMessage`: Encapsula variáveis dinâmicas do cliente web.
- Classe `AssistantMessage`: Mantém o histórico stateless da conversa na aplicação servidora.

Configuração Declarativa (application.properties)

O Spring Boot abandona códigos de configuração espalhados por *properties* centralizados.

```
# src/main/resources/application.properties

# Ativando a propriedade da chave globalmente
spring.ai.openai.api-key=${OPENAI_API_KEY}

# Padronizando o modelo que todas as interfaces do projeto usarao
spring.ai.openai.chat.options.model=gpt-4o-mini
spring.ai.openai.chat.options.temperature=0.2
```

A sintaxe ${...} busca dinamicamente no sistema operacional ou no arquivo .env acoplado à IDE.

Segurança Corporativa de Credenciais no Spring

Diferente de scripts ingênuos em Python que usam `load_dotenv()`, um microsserviço Spring Boot habitualmente é implantado num contêiner *Docker* em um cluster *Kubernetes*.

- O ambiente orquestrador injetará a credencial secretamente e em tempo de execução na RAM do contêiner.
- O código fonte Java jamais manipulará, gravará em disco ou fará *logs* (`System.out.println`) das *API Keys*. Elas trafegam estritamente da propriedade injetada para o encabeçalho HTTP da biblioteca subjacente.

Portabilidade Absoluta (Zero Code Change)

Para migrar de OpenAI para Gemini 1.5 e cortar os custos do *backend* pela metade, o time de engenharia precisa fazer apenas **duas coisas**:

1. Trocar a dependência do `pom.xml` de `spring-ai-openai-boot-starter` para `spring-ai-vertex-ai-gemini-spring-boot-starter`.
2. Atualizar a chave no `application.properties`.

A classe Java contendo a injeção (`private final ChatClient chatClient;`) e toda a lógica de *prompts* não exige a mudança de **um único caractere**.

Inicializando um Projeto (Spring Initializr)

- O padrão inicial não é começar do zero, mas visitar o portal `start.spring.io` (ou usar o plugin da IDE).
- Escolhe-se o Gerenciador (Maven), a versão do Java (geralmente Java 17 ou 21 LTS).
- Adiciona-se a dependência web (*Spring Web*) para expor a IA como uma API REST na porta 8080.
- Adiciona-se o provedor oficial (*Spring AI - OpenAI* ou similar).

Implementação: O Controlador REST

Unindo a engenharia Web tradicional com a IA Generativa.

```
RestControllerRequestMapping("/api/chat")
public class ChatController {

    private final ChatClient chatClient;

    // Inversao de Controle: O Spring injeta o motor de IA
       configurado
    public ChatController(ChatClient.Builder builder) {
        this.chatClient = builder.build();
    }
}
```

Implementação: O Endpoint Generativo

```
    GetMapping("/gerar")public String
        gerarResposta(RequestParam String duvida) {
         return this.chatClient.prompt()
                 .system("Voce e o assistente da disciplina de IA.")
                 .user(duvida)
                 .call()
                 .content();
    }
    // Uma requisicao HTTP GET http://localhost:8080/api/chat/gerar?
        duvida=O que e RAG?
    // devolvera uma string pura computada na nuvem.
```

Structured Outputs: De LLM para Java Records

Um dos desafios clássicos é o mapeamento objeto-relacional ou mapeamento de dados. O Spring AI resolve o *JSON Mode* (visto na Aula 16) brilhantemente usando *Java Records* nativos e o `BeanOutputConverter`.

- O desenvolvedor tipa a saída exigida: `public record Analise(String categoria, boolean urgente) {}`.
- O Spring AI varre a reflexão da classe (*Reflection*), converte para *JSON Schema*, injeta ocultamente no *System Prompt* da OpenAI, recebe a *String JSON* limpa e mapeia (*Deserialization*) automaticamente para o objeto Java.

O Mapeamento Objeto-Vetor (O que vem a seguir?)

Da mesma forma que o JPA/Hibernate mapeou os bancos de dados relacionais para classes Java (ORM), o Spring AI propõe um mapeamento para os **Vector Databases**.

- Com o ecossistema montado, as próximas aulas usarão a interface `Document` e as interfaces abstratas do `VectorStore` para reconstruirmos o poderoso RAG, mas desta vez imersos num ecossistema de altíssima coesão estrutural e preparado para produção industrial.

Conclusão da Aula 18

- Transicionar para a arquitetura Java/Spring Boot representa o amadurecimento das soluções em Inteligência Artificial Generativa do laboratório (*Jupyter Notebook*) para as linhas de frente corporativas globais (*Produção*).
- Padrões de *design* robustos como Inversão de Controle, *Builders* fluentes e interfaces padronizadas garantem a portabilidade em um ecossistema volátil.
- O *Spring AI* blinda a aplicação contra o efeito de *Vendor Lock-in*, permitindo evoluções de modelo ágeis, tipagem profunda (*Records*) e implantação altamente paralelizável e resiliente a falhas.

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: Spring AI & FastAPI

100% da Aula 18 Concluída!