Inteligência Artificial 2

Aula 22: Projeto Prático RAG com Spring

Prof. Aléssio Miranda Júnior
alessio@cefetmg.br

CEFET-MG - Campus Timoteo
Dep. Engenharia de Computação

Inteligência Artificial 2

Sumário

Onde Paramos?

Na aula anterior, adicionamos **resiliência** ao sistema:

- `@Retryable` com Exponential Backoff (2s, 4s).
- `@Recover` para fallback amigável ao usuário.
- Separação Controller → Service (arquitetura limpa).

A Missão de Hoje

Hoje é **dia de integração**. Todas as peças do TP2 devem se conectar em um produto funcional completo.

Objetivos da Sessão

Ao final desta aula, a dupla deve ter:

1. **Ingestão** de PDFs reais no VectorStore (pipeline ETL rodando).
2. **Endpoint RAG** funcional respondendo perguntas fundamentadas.
3. **Resiliência** (`@Retryable` + fallback) validada.
4. **Front-end** conectado ao back-end via CORS.
5. **Segurança**: chaves de API fora do código (`.env`).

Da Prova de Conceito à Produção

Chegamos ao último nível da disciplina.

- Já compreendemos os fundamentos matemáticos (vetores, transformadas) e experimentamos APIs de IA em scripts isolados.
- Agora, vamos empacotar um sistema de IA RAG dentro de um produto real: um microsserviço Java Spring Boot, estruturado para suportar milhões de requisições, rodando de forma isolada através de Contêineres.
- Esta é a **Engenharia de IA**: criar infraestrutura escalável ao redor de redes neurais.

Arquitetura Limpa e Ports and Adapters

- Num sistema corporativo, não misturamos acesso a banco de dados, chamada de API do Google e lógica de negócios na mesma classe.
- Utilizamos a **Arquitetura Hexagonal**:
  1. **Domínio (Centro):** As regras que definem o seu negócio.
  2. **Portas (Interfaces):** Contratos de como a aplicação se comunica com o exterior (ex: *VectorStore*).
  3. **Adaptadores (Bordas):** Implementações concretas (REST Controllers para a Web, *PgVector* para o Banco).

Por que PostgreSQL com `pgvector`?

- Durante o curso usamos o ChromaDB em memória. Excelente para estudo, mas frágil para alta disponibilidade.
- O **PostgreSQL** é o banco de dados relacional *Open Source* mais confiável do mercado. A extensão `pgvector` permite que ele armazene dados relacionais tradicionais e Tensores Densos na mesma tabela.
- Vantagem Corporativa: Reutilizar a infraestrutura existente (*RDS na AWS, backups já configurados*) sem precisar de um novo banco exótico.

Infraestrutura Ágil: Docker Compose

O *Docker Compose* permite que você e todos os desenvolvedores do seu time subam o banco vetorial com um comando (`docker-compose up`).

```
services:
  pgvector:
    image: pgvector/pgvector:pg16
    environment:
      POSTGRES_USER: usuario
      POSTGRES_PASSWORD: password
      POSTGRES_DB: base_rag
    ports:
      - "5432:5432"
```

Configurando o `application.yml`

Com o banco rodando localmente, o Spring precisa de credenciais duplas: as chaves da LLM e as de acesso ao banco vetorial.

```
spring:
  ai:
    openai:
      api-key: ${OPENAI_API_KEY}
    vectorstore:
      pgvector:
        index-type: HNSW
        distance-type: COSINE_DISTANCE
        dimensions: 1536
  datasource:
    url: jdbc:postgresql://localhost:5432/base_rag
```

Migrações de Banco (Flyway ou Spring Init)

- A tabela `vector_store` precisa existir no PostgreSQL.
- O Spring AI (quando acoplado ao PgVector) possui uma propriedade para inicializar e criar a tabela de *Embeddings* automaticamente.
- Na tabela `vector_store`, haverá colunas para o identificador (`id` UUID), conteúdo literal (`content` TEXT), metadados dinâmicos (`metadata` JSONB) e o vetor em si (`embedding` VECTOR).

Ingestão e Carga (O Controlador Admin)

Precisamos de uma Rota Oculta que o *Admin* chama para alimentar o cérebro da empresa.

```
@PostMapping("/admin/ingestar")
public String ingestarPDF(@RequestParam("file") MultipartFile file) {
    // 1. Converter file para Resource
    var reader = new PagePdfDocumentReader(resource);
    // 2. Fragmentar em blocos menores (Overlap)
    var chunks = new TokenTextSplitter().apply(reader.get());
    // 3. VectorStore abstrai e injeta no PostgreSQL automaticamente
    vectorStore.add(chunks);
    return "Documento vetorizado com sucesso.";
}
```

O RAG Service: Coração Cognitivo

O *Service* concentra a inteligência da busca e conversação.

```
@Service
public class ChatService {
    private final ChatClient chatClient;

    // Injeta o VectorStore via QuestionAnswerAdvisor
    public ChatService(ChatClient.Builder builder, VectorStore store) {
        this.chatClient = builder
            .defaultAdvisors(new QuestionAnswerAdvisor(store))
            .build();
    }

    public String responder(String pergunta) {
        return chatClient.prompt().user(pergunta).call().content();
    }
}
```

Paginação e Filtros de Metadados

O RAG avançado vai além da pura similaridade de cossenos.

- Muitas vezes você só quer buscar um vetor se ele pertencer ao departamento jurídico. Para isso usamos **Metadata Filtering**.
- A consulta vetorial é modificada em tempo real pelo *Advisor* para adicionar cláusulas SQL:
- `SELECT * FROM vector_store WHERE embedding <=> [0.12...] AND metadata->>'departamento' = 'juridico' LIMIT 5.`

O Controlador REST: Expondo a API

- O seu microsserviço precisa expor a infraestrutura em rotas HTTP acessíveis via requisições GET ou POST.
- Uma boa prática em APIs *Enterprise* é não receber nem enviar texto bruto (Strings), e sim utilizar "Data Transfer Objects" (DTOs).
- No Java, utilizamos a classe `record` para definir esses contratos imutáveis (Ex: `ChatRequest` e `ChatResponse`).

Contratos de API (Records)

```
// DTOs garantem que o JSON enviado pelo Frontend sera estruturado.
public record ChatRequest(
    @NotBlank String mensagemUsuario,
    @NotNull String idSessao
) {}

public record ChatResponse(
    String respostaDaIA,
    List<String> fontesCitadas,
    int duracaoEmMs
) {}
```

O Controlador REST Completo

```
@RestController
@RequestMapping("/api/v1/ia")
@CrossOrigin(origins = "*") // Libera o navegador Frontend
public class IaController {

    private final ChatService service;

    @PostMapping("/perguntar")
    public ChatResponse perguntar(@RequestBody ChatRequest req) {
        long inicio = System.currentTimeMillis();
        String resp = service.responder(req.mensagemUsuario());
        return new ChatResponse(resp, null, System.currentTimeMillis() - inici
    }
}
```

Tratamento Global de Exceções (`@ControllerAdvice`)

E se o ChromaDB cair? E se a cota de dólares da OpenAI acabar (Erro 429)? O usuário não pode ver o erro "HTTP 500 Internal Server Error" genérico.

```
@RestControllerAdvice
public class TratadorErroGlobal {

    @ExceptionHandler(OpenAiHttpException.class)
    public ResponseEntity<ErroDTO> falhaOpenAI(Exception e) {
        return ResponseEntity.status(503).body(
            new ErroDTO("Servico Cognitivo Indisponivel no momento")
        );
    }
}
```

Segurança: Limite de Taxa (Rate Limiting)

- Requisições para APIs de Inteligência Artificial custam dinheiro vivo (por conta da tarifa de *Tokens* gerados).
- Sem limite de taxa, um usuário com um script (*loop* infinito) pode causar um custo massivo na sua nuvem AWS/OpenAI.
- Uma infraestrutura de RAG corporativa exige a configuração de *Rate Limiters* (Ex: *Resilience4J* ou *Redis Token Bucket*), limitando cada IP a 10 requisições por minuto.

Validando a API (Postman e Swagger)

- Antes de escrever qualquer tela gráfica (Front-end), você deve testar o fluxo de ponta a ponta.
- Utilize ferramentas como **Postman** ou **Insomnia** para realizar um `POST /api/v1/ia/perguntar`.
- Alternativamente, habilite a dependência `springdoc-openapi` para gerar documentação interativa e botões de teste automaticamente através do **Swagger UI** na URL da sua aplicação.

Testes de Integração em IAs

- Como testar um algoritmo cujo texto resultante varia probabilisticamente a cada execução?
- Testes de RAG focam na Etapa de Recuperação: *"Dado o Documento A, o VectorStore retornou-o no Top 3?"*
- A camada de LLM é **mockada** (simulada) nos testes de Integração usando bibliotecas padrão como Mockito, retornando strings constantes para comprovar que o controlador e a rede estão funcionando.

Subindo Bancos Dinâmicos com Testcontainers

- Um erro amador é testar o código no seu banco Postgres local e assumir que "na minha máquina funciona".
- **Testcontainers:** Biblioteca Java que sobe automaticamente um contêiner *Docker* de PgVector toda vez que você roda a suíte do *JUnit*.
- Ao final do teste unitário, o contêiner morre, deixando o ambiente perfeitamente limpo para os *Pipelines* (CI/CD) do GitHub Actions.

Escalabilidade do Microsserviço de RAG

- O Spring Boot é *Stateless* (não guarda estado local). A memória conversacional fica no Banco de Dados (Redis ou Postgres).
- Se o site recebe repentinamente tráfego de Natal (*Black Friday*), orquestradores de Nuvem (Kubernetes/EKS) replicam sua aplicação de 1 instância para 50 réplicas instáveis.
- Essa replicação lida perfeitamente com 50.000 requisições por segundo. O único gargalo passa a ser o provedor da Inteligência Artificial em si, ou a cota monetária estipulada pela diretoria.

Encerramento: O Futuro do Desenvolvedor em IA

Chegamos ao fim da trilha do Arco 3.

- O papel do Engenheiro de Software não foi extinto pelos Modelos de Linguagem; foi **elevado**.
- Passamos da criação de rotinas "If-Else" procedurais pesadas para a orquestração e gerenciamento de microsserviços *cognitivos*.
- O diferencial no mercado de trabalho agora não é apenas "saber programar algoritmos", mas sim arquitetar fluxos assíncronos e robustos capazes de "aprender", consultar documentos gigantescos nativamente com *Embeddings* e executar funções seguras. O futuro pertence a arquitetos de sistemas inteligentes!

Inteligência Artificial 2

O conhecimento adquirido em RAG, Engenharia de Prompt e Infraestrutura Vetorial com *Spring AI* habilita você a construir sistemas *Enterprise* complexos e inovações que transformarão o fluxo de dados global nas próximas décadas.

Parabéns e sucesso na sua jornada!

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: Fine-Tuning (LoRA)

100% da Aula 22 Concluída!