Session Summary: RAG with Spring AI and Langchain4j using Postgres with pgvector

This document summarizes what we learned during a proof of concept session on building a retrieval-augmented generation (RAG) pipeline in Java. The goal was to ingest documents, index them in a vector database, and use a large language model to answer questions with context from those documents. Our stack used Spring AI with langchain4j and langgraph4j, and Postgres with the pgvector extension for storing embeddings. Along the way we hit several roadblocks and solved them.

Ingest and Index Pipeline

We developed an ingestion pipeline that reads documents from disk, splits them into chunks, enriches each chunk with metadata, embeds them and writes them into a Postgres table. The pipeline consists of:

• A parser, such as Apache PDFBox, to convert PDFs into langchain4j Document objects. We used stable version 0.36.2.
• A chunker built from DocumentByParagraphSplitter with a DocumentBySentenceSplitter fallback. Paragraph splitting preserves structure and sentences handle large paragraphs. It takes a maxCharacters and maxOverlap parameter.
• A SegmentMapper that converts each TextSegment into a Spring AI Document, copying metadata and adding a chunk_id (either deterministic hash or random UUID) and other metadata like source and title.
• A Spring AI VectorStore configured for Postgres with pgvector (rag_chunks table). We specified the number of dimensions to match our embeddings (e.g., 384 for MiniLM) and added JSONB and full-text indices.

We emphasised adding metadata keys such as doc_id, doc_type, effective_date, version and section so we can filter and attribute results later.

Database Setup and Gotchas

Postgres does not include pgvector by default. We had to install the correct package for our OS (postgresql-14-pgvector) from the PGDG repository. We solved a repository error by using the jammy (Ubuntu 22.04) codename instead of mint's victoria. After installation we created the extension in the correct database and schema with CREATE EXTENSION vector.

Next we created the rag_chunks table with columns:

• id BIGSERIAL PRIMARY KEY
• content TEXT NOT NULL
• metadata JSONB
• embedding VECTOR(384)

We added GIN index on metadata, GIN index on a tsvector column for full-text search and an HNSW index on embedding. We also ensured the search_path included the schema containing the vector type. Later we added a generated fts column (to_tsvector('english', content)).

Other challenges included logging in as the right database user, resetting the postgres user password and using psql with -U and -d flags. We created a dedicated app role and granted USAGE and CREATE on the schema. Finally, we avoided using "var" or alias imports in Java which are not valid.

Retrieval and Langgraph Integration

We wrote a RagRetriever service that queries the vector store using similaritySearch and optional metadata filters. We plan to extend it with hybrid retrieval by querying full-text indices and merging results before reranking.

On top of the retrieval service we defined a graph in langgraph4j to orchestrate conversations. The graph routes user questions through a tool-routing node, runs application-specific tools if necessary, then calls the retrieve node to fetch context documents. The synthesize node assembles a prompt with the question, tool outputs and context. The assistant node (LLM) generates an answer which is validated to ensure citations or "I don't know" are present.

We emphasised the importance of deterministic control flow and error handling in the graph. Langgraph4j lets us retry or branch depending on validation results.

Gotchas Encountered and Lessons Learned

• Spring AI starter names changed: we used spring-ai-starter-model-openai and spring-ai-starter-vector-store-pgvector instead of older names. For langchain4j we used langchain4j-document-parser-apache-pdfbox and the sentence/paragraph splitters instead of RecursiveCharacterTextSplitter (which only exists in Python).
• Document classes differ: langchain4j Document is not the same as org.springframework.ai.document.Document. We mapped TextSegment to Spring AI Document manually.
• We fixed a wrong apt repository (victoria-pgdg) and installed pgvector for PostgreSQL 14. After installation we had to create the extension per database and ensure search_path includes the extension's schema.
• We normalised embedding dimension (384) to match our chosen model and updated table definition and Spring AI config accordingly.
• Logging in as the wrong user caused type vector not found errors. We switched to the target database and created the extension there.
• We avoided using Java's var alias imports because Java does not support aliasing in import statements.

These lessons underscore the importance of matching versions, understanding library differences between Python and Java, and checking the environment when errors occur.