LangChain4j with Spring Boot

Author: Konrad Kowalczyk
Repository: [xkondix AI Sandbox]

Presentation Plan

Introduction

A **Large Language Model (LLM)** is an advanced AI model trained on massive amounts of text data to understand and generate human-like language.

It can answer questions, write code, summarize documents, translate languages, and much more.

Token is a basic unit of text (like a word or part of a word) that language models use to process and generate responses.

The purpose of this presentation and project is to demonstrate how you can easily integrate **Spring Boot** with **LLM** using the **LangChain4j** library.

Configuration

1. Project Setup
2. LangChain4j dependencies
3. Docker compose

Project Setup

- Spring Boot 3.5.3 (with LangChain4j)
- JAVA 21
- Ollama for local LLMs
- Redis for memory
- ChromaDB for vector store

LangChain4j dependencies

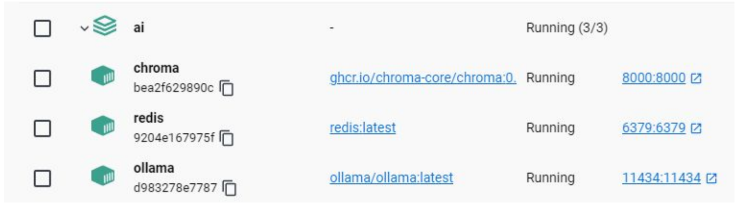
Useful dependencies (more in project):

- langchain4j & langchain4j-core (core)
- langchain4j-open-ai (gpt api)
- langchain4j-ollama (local LLMs)
- langchain4j-embeddings (models converting text -> vectors)
- langchain4j-embeddings-all-minilm-l6-v2-q (model used in project)
- langchain4j-chroma (chromaDB used as a vector database in the project)
- langchain4j-redis (used as a permanent chat memory in the project)

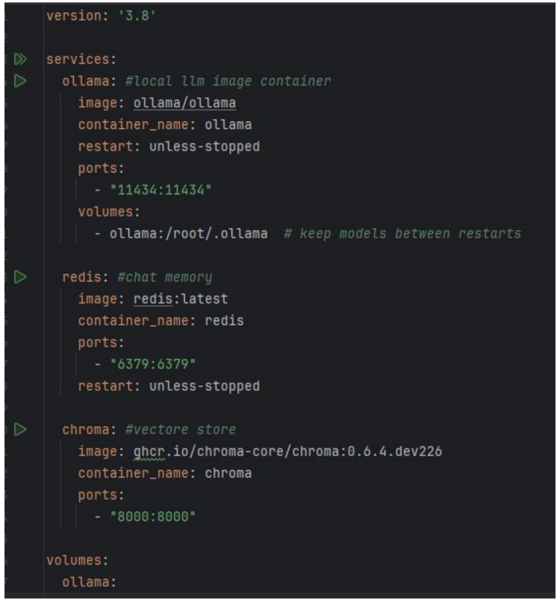
Docker compose

Steps required to download LLMs:

```
1.   docker compose up -d
2.   docker exec -it ollama bash
3.   ollama pull gemma3:4b
4.   ollama pull llama3.1:8b
```



ai		-	Running (3/3)	
chroma	bea2f629890c	ghcr.io/chroma-core/chroma:0.	Running	8000:8000
redis	9204e167975f	redis:latest	Running	6379:6379
ollama	d983278e7787	ollama/ollama:latest	Running	11434:11434



```yaml
version: '3.8'

services:
  ollama: #local llm image container
    image: ollama/ollama
    container_name: ollama
    restart: unless-stopped
    ports:
      - "11434:11434"
    volumes:
      - ollama:/root/.ollama  # keep models between restarts

  redis: #chat memory
    image: redis:latest
    container_name: redis
    ports:
      - "6379:6379"
    restart: unless-stopped

  chroma: #vectore store
    image: ghcr.io/chroma-core/chroma:0.6.4.dev226
    container_name: chroma
    ports:
      - "8000:8000"

volumes:
  ollama:
```

Integration with LLMs

1. Modal Parameters
2. Local (gemma3 & llama3.1)
3. API (OpenAI chat GPT)

Model Parameters

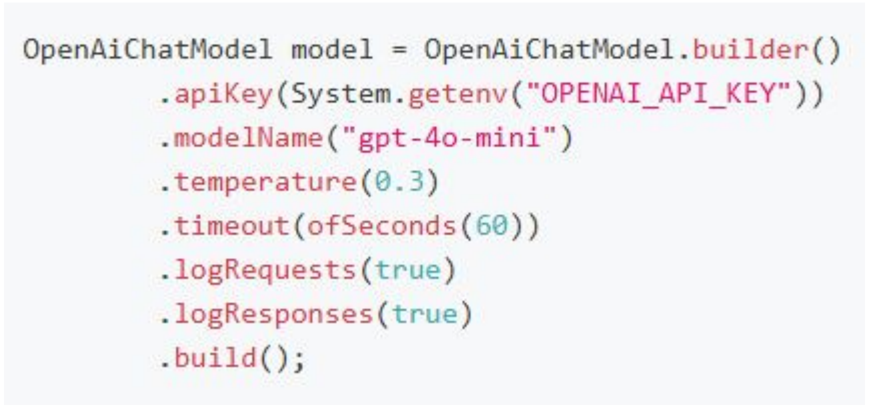
`modelName` the name of the model (e.g. gemma3:4b).

`temperature` **(0–2)**

- Higher = more random
- Lower = more deterministic

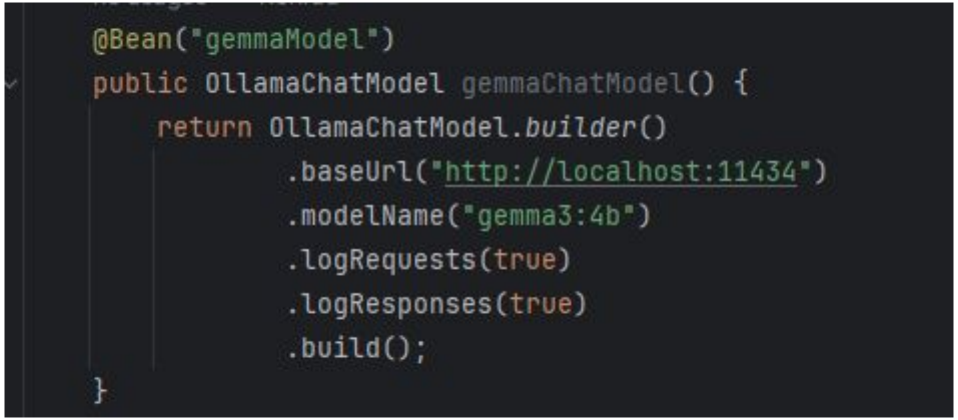
`maxTokens` the maximum number of tokens that can be generated in the chat completion.

more here doc



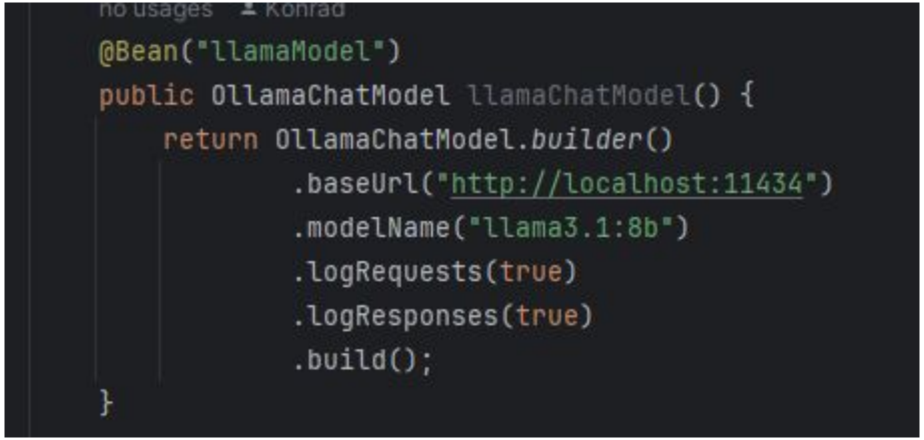
```
OpenAiChatModel model = OpenAiChatModel.builder()
        .apiKey(System.getenv("OPENAI_API_KEY"))
        .modelName("gpt-4o-mini")
        .temperature(0.3)
        .timeout(ofSeconds(60))
        .logRequests(true)
        .logResponses(true)
        .build();
```

Local (gemma3)



```
@Bean("gemmaModel")
public OllamaChatModel gemmaChatModel() {
    return OllamaChatModel.builder()
            .baseUrl("http://localhost:11434")
            .modelName("gemma3:4b")
            .logRequests(true)
            .logResponses(true)
            .build();
}
```

Local (llama 3.1)

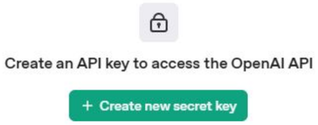


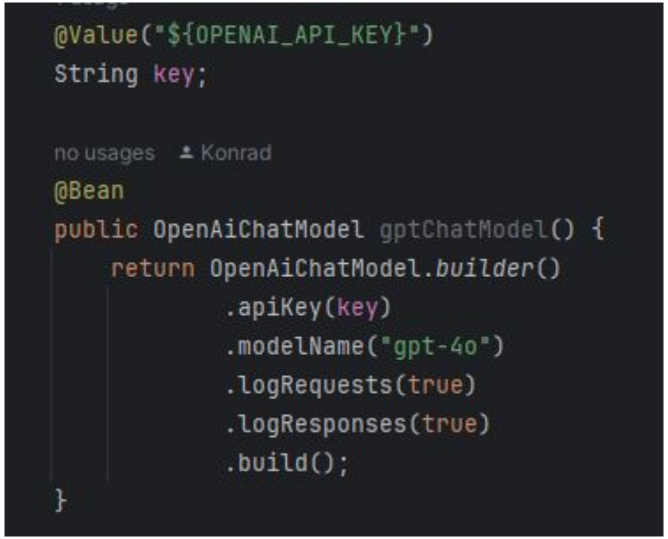
```
no usages  Konrad
@Bean("llamaModel")
public OllamaChatModel llamaChatModel() {
    return OllamaChatModel.builder()
            .baseUrl("http://localhost:11434")
            .modelName("llama3.1:8b")
            .logRequests(true)
            .logResponses(true)
            .build();
}
```

API OpenAI chat GPT

Using ChatGPT in your application requires an OpenAI API key and specifying the desired model (like gpt-4) when making requests to the API.

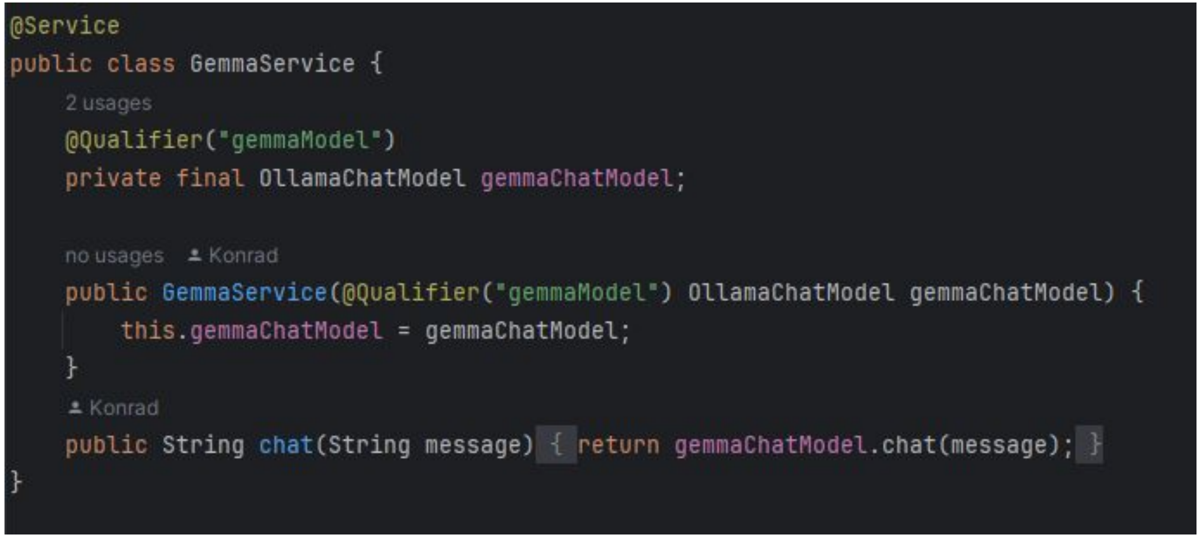
Link to create a key GPT key


Create an API key to access the OpenAI API



```
@Value("${OPENAI_API_KEY}")
String key;

no usages  Konrad
@Bean
public OpenAiChatModel gptChatModel() {
    return OpenAiChatModel.builder()
            .apiKey(key)
            .modelName("gpt-4o")
            .logRequests(true)
            .logResponses(true)
            .build();
}
```

Chat



```
@Service
public class GemmaService {
    2 usages
    @Qualifier("gemmaModel")
    private final OllamaChatModel gemmaChatModel;

    no usages   Konrad
    public GemmaService(@Qualifier("gemmaModel") OllamaChatModel gemmaChatModel) {
        this.gemmaChatModel = gemmaChatModel;
    }
    Konrad
    public String chat(String message) { return gemmaChatModel.chat(message); }
}
```

Chat with memory

1. Before (AiServices)
2. Temporary Memory
3. Permanent Memory (redis)

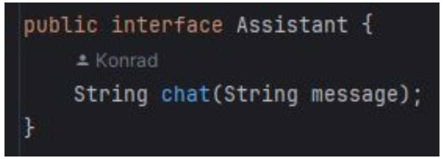
AiServices

AI Services in LangChain4j provide a declarative interface to interact with LLMs.

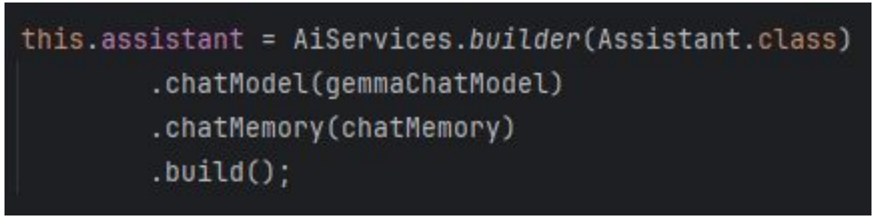
Models, hiding the complexity behind simple, annotated methods.

It supports features such as:

- Memory
- Tools
- RAG



```java
public interface Assistant {
    Konrad
    String chat(String message);
}
```

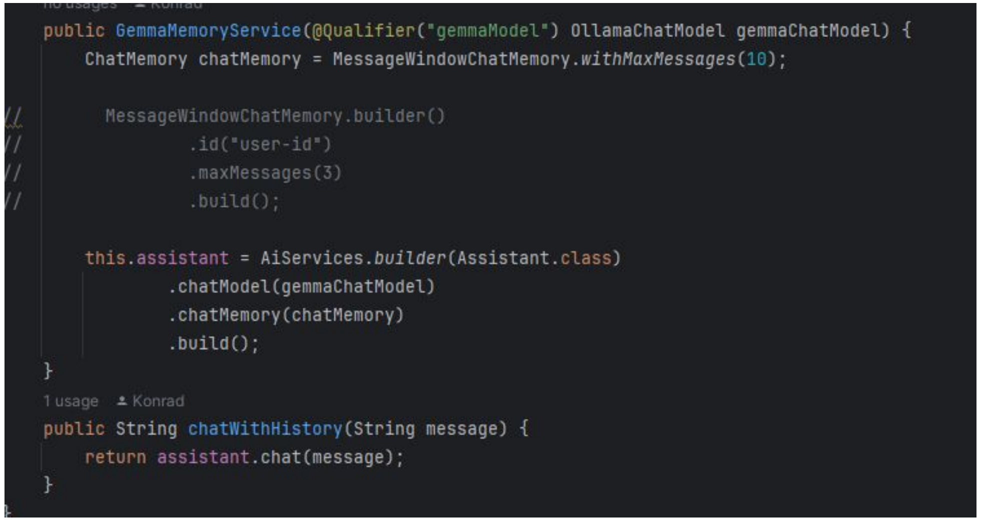


```java
this.assistant = AiServices.builder(Assistant.class)
        .chatModel(gemmaChatModel)
        .chatMemory(chatMemory)
        .build();
```

Temporary memory

There are two common approaches for temporary memory:

- with user
- without user

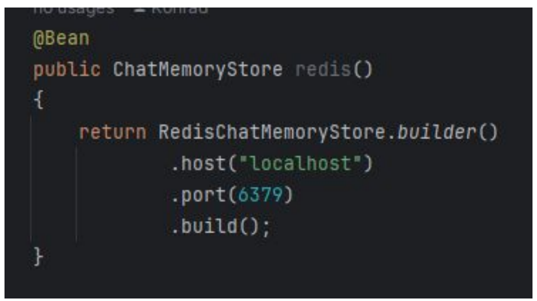


```
no usages  Konrad
public GemmaMemoryService(@Qualifier("gemmaModel") OllamaChatModel gemmaChatModel) {
    ChatMemory chatMemory = MessageWindowChatMemory.withMaxMessages(10);

//      MessageWindowChatMemory.builder()
//              .id("user-id")
//              .maxMessages(3)
//              .build();

    this.assistant = AiServices.builder(Assistant.class)
            .chatModel(gemmaChatModel)
            .chatMemory(chatMemory)
            .build();
}
1 usage  Konrad
public String chatWithHistory(String message) {
    return assistant.chat(message);
}
```

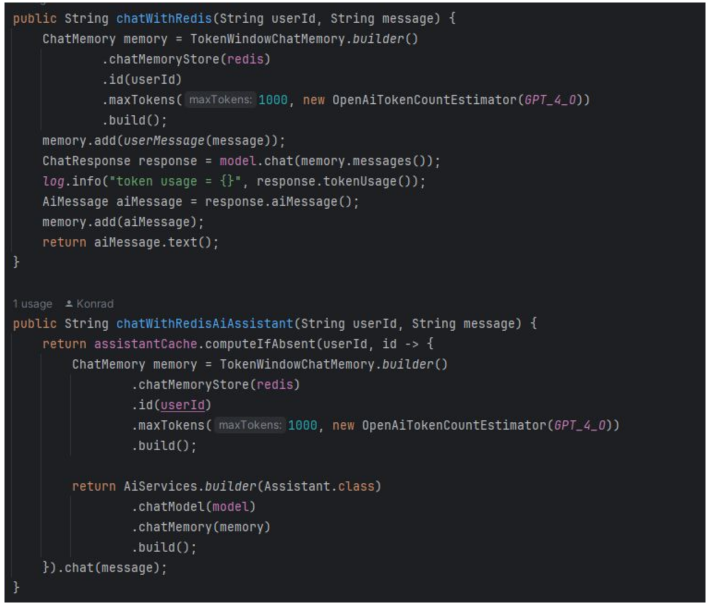
Permanent Memory (redis)

Redis is an in-memory data store that we can use to persist chat memory across sessions.

Before using it, we need to configure the Redis database and integrate it with our application (as shown in the setup screenshots).



```
no usages   Konrad
@Bean
public ChatMemoryStore redis()
{
    return RedisChatMemoryStore.builder()
            .host("localhost")
            .port(6379)
            .build();
}
```



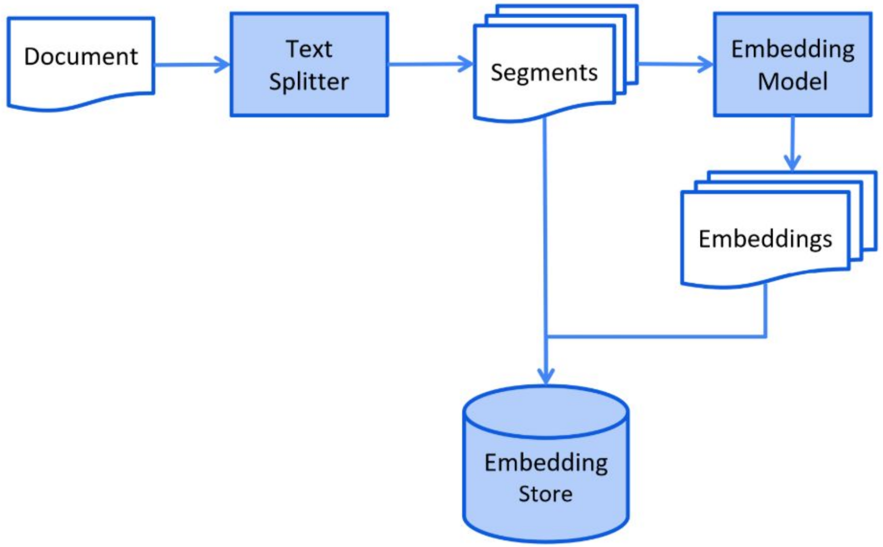
```
public String chatWithRedis(String userId, String message) {
    ChatMemory memory = TokenWindowChatMemory.builder()
            .chatMemoryStore(redis)
            .id(userId)
            .maxTokens( maxTokens: 1000, new OpenAiTokenCountEstimator(GPT_4_O))
            .build();
    memory.add(userMessage(message));
    ChatResponse response = model.chat(memory.messages());
    log.info("token usage = {}", response.tokenUsage());
    AiMessage aiMessage = response.aiMessage();
    memory.add(aiMessage);
    return aiMessage.text();
}

1 usage   Konrad
public String chatWithRedisAiAssistant(String userId, String message) {
    return assistantCache.computeIfAbsent(userId, id -> {
        ChatMemory memory = TokenWindowChatMemory.builder()
                .chatMemoryStore(redis)
                .id(userId)
                .maxTokens( maxTokens: 1000, new OpenAiTokenCountEstimator(GPT_4_O))
                .build();

        return AiServices.builder(Assistant.class)
                .chatModel(model)
                .chatMemory(memory)
                .build();
    }).chat(message);
}
```

Retrieval-Augmented Generation (RAG)

Retrieval-Augmented Generation (RAG) is a technique that improves language models by retrieving relevant external information to generate more accurate and context-aware responses.

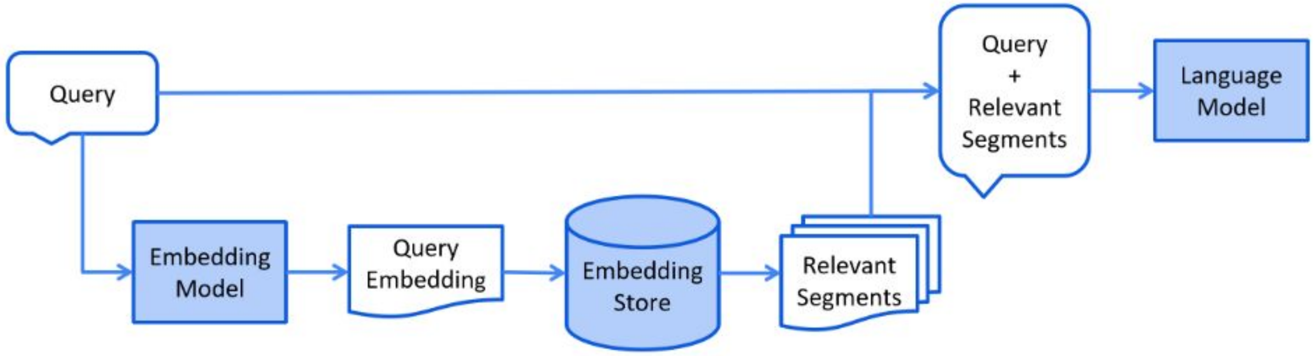
1. Documents Search
2. Web Search
3. Database Search

Retrieval-Augmented Generation (RAG)



source: https://docs.langchain4j.dev/tutorials/rag

Retrieval-Augmented Generation (RAG)



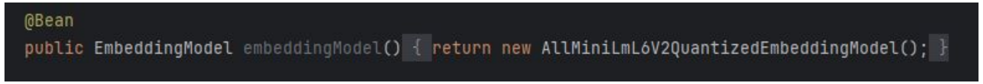
source: https://docs.langchain4j.dev/tutorials/rag

Documents Search

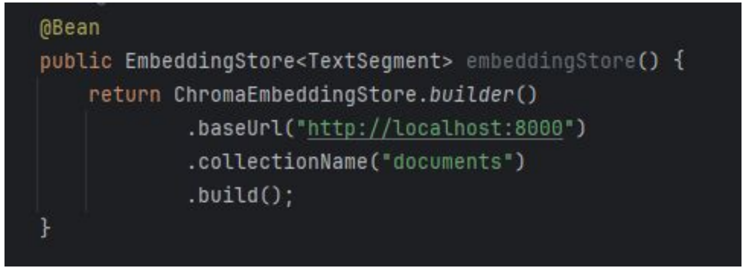
The provided configuration in the project is adapted specifically for document-based RAG. It uses Chroma vector storage and embedding model to process and retrieve relevant text segments from uploaded files.

Source:

- Local documents
- PDF files
- Texts



```java
@Bean
public EmbeddingModel embeddingModel() { return new AllMiniLmL6V2QuantizedEmbeddingModel(); }
```



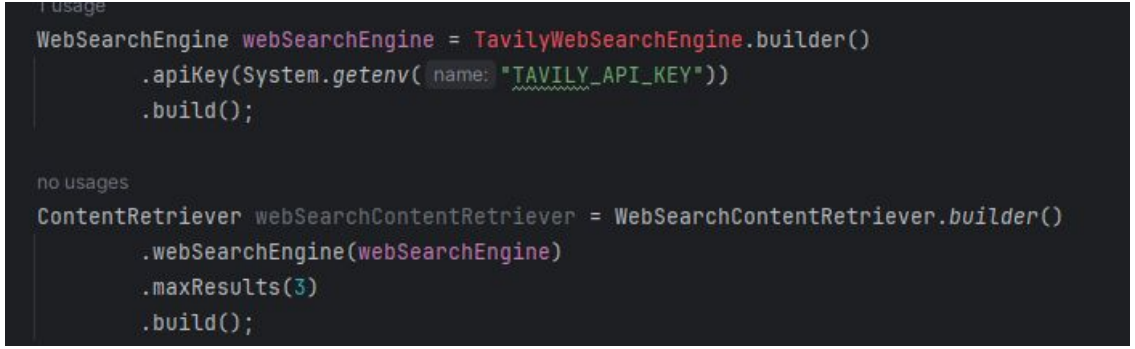
```java
@Bean
public EmbeddingStore<TextSegment> embeddingStore() {
    return ChromaEmbeddingStore.builder()
            .baseUrl("http://localhost:8000")
            .collectionName("documents")
            .build();
}
```

Web Search

Before you start, you need to create an account at [Tavily platform]() to obtain an API key.

You can also choose a different web search engine, but this one is recommended.

Example: [Web Search Example]()



```
1 usage
WebSearchEngine webSearchEngine = TavilyWebSearchEngine.builder()
        .apiKey(System.getenv( name: "TAVILY_API_KEY"))
        .build();

no usages
ContentRetriever webSearchContentRetriever = WebSearchContentRetriever.builder()
        .webSearchEngine(webSearchEngine)
        .maxResults(3)
        .build();
```

Database Search

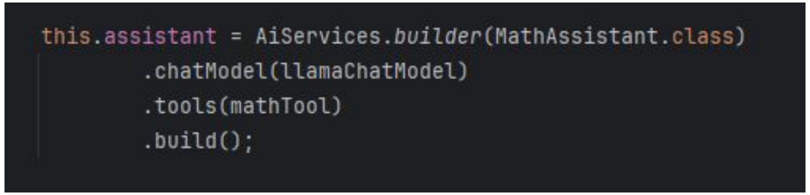
Before you start, you need create connection to database.

Example: Database Search Example

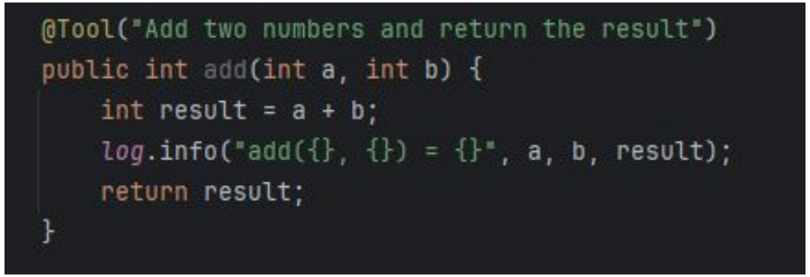
Tools (function calling)

Tools let an LLM indicate when it needs to use an external function or API by requesting a specific tool call with arguments instead of answering directly.

This means that we can provide a function written in JAVA with an appropriate description that will be called to get the answer, such as the add method as in the example.



```java
this.assistant = AiServices.builder(MathAssistant.class)
        .chatModel(llamaChatModel)
        .tools(mathTool)
        .build();
```



```java
@Tool("Add two numbers and return the result")
public int add(int a, int b) {
    int result = a + b;
    log.info("add({}, {}) = {}", a, b, result);
    return result;
}
```

Model Context Protocol (MCP)

Model Context Protocol (MCP) is a standard that allows LLM to communicate with external MCP-compliant servers that provide and execute tools via HTTP or stdio protocols.

LangChain4j supports MCP, allowing clients to connect to these servers, download available tools and use them during AI interactions, enabling seamless integration of external functions into AI workflows.

Example: MCP Example

Questions?
Thanks for your time!