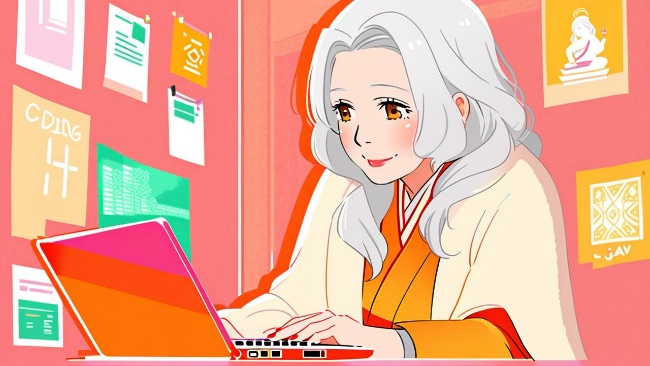
RAG apps for legacy spring developers

Introduction to RAG apps and review of the new Spring AI library

The new Spring AI library is about to be released in version 1.0, what better opportunity to present the operating of RAG applications to us, legacy Java developers?

If you've been using Spring for a lifetime but don't know anything about LLM, this review will give you a comprehensible explanation of how a chatbot, summarizing or classifying documents, and querying private documents applications works.

In this field, the langchain python library is the master, so it will be my concern to systematically compare it with the implementation choices of Spring AI.

Italian prompts can, perhaps, be appreciated, to show a localized use of the famous ChatGPT.

As usual, the source code of the project is available on github and you will find a long annotated bibliography.

Related Articles:

* [Local code llama with gradio and hugging face](https://medium.com/@nicolasanti_43152/local-code-llama-with-gradio-and-hugging-face-1153112046ec?source=your_stories_page-------------------------------------)
* [Bootstrap a LLaMa assistant on personal data](https://medium.com/@nicolasanti_43152/bootstrap-a-llama-assistant-on-personal-data-16062fa5aa6d?source=your_stories_page-------------------------------------)

# A SQL analogy

Our legacy applications interface with those peculiar external services that are relational databases using simple structured text in SQL language. Interaction with LLMs is also configured as a call to services intermediated by plain text, in this case called **prompt**, structured in natural language.

In the absence of an Obecjt to Relation Mapping (ORM) library such as JPA or Hibernate, the most common solution to access a database is to delegate the composition of SQL to a class using some string templates. Class and template are placed in the layer called persistence (also known as DAO) which has the task of abstracting the details of connection and communication with a specific database to the rest of the application.

select {fields\_list} from {tables\_list} where 1=1 {where\_clauses}

Abstraction is also made possible using the so-called drivers (odbc, jdbc): code developed directly from the database producer (e.g. postgress, oracle, mysql).

A screenshot of a computer

Description automatically generated

The same is true for a language application that, using text templates, must produce understandable prompts for a specific LLM and take care of the connection and communication with them, abstracting the details from the rest of the application.

You are a PM of a software house. Write a summary of {max\_words} words maximum based on the following project charter: {synopsis}

Here too, communication is made possible using code developed by the manufacturer of the AI model (llama, openai). Over these, there are further layers of software to access different LLMs in the same way, abstractions provided by libraries such as langchain.

# Completion

The use of an LLM, therefore, in its simplest form, takes the form of a call to a web service and, among the application layers, is placed at the same height as the repositories (aka DAO). Please, note that the layer would not change if the database was running locally, e.g. *H2DB*, or the LLM was delivered locally, e.g. via *vllm* or *llama.cpp*.

In the *langchain* library, the family of classes that interfacing with an LLM are called Chain, because they are designed to be linked to each other. They require, in the most basic form, two elements: an LLM model and templates for the prompt. In the Spring AI library, the equivalent class is called ChatClient and requires the same elements: a model derived from ChatModel and templates derived from PromptTemplate.

Calls to the LLM, think of a chat, must contain at least two types of messages, each relating to a participant or, as it says, to a **role**:

* **USER:** these are the messages entered by the human being interacting with the AI;
* **ASSISTANT:** is the AI's answer;

To these are added two, perhaps less intuitive ones which are **SYSTEM** and **TOOL,** the latter we will talk about later. Messages with the SYSTEM role provide general guidance to the model. For example,:

You are a polite and willing assistant to a software house. Your job is to respond politely to customer requests, trying to make yourself useful, fully understand the customer's problem and collect all the information necessary to open the eventual ticket.  
  
The information required is:  
\* the customer number{more\_infos}

Notes:

* Above we see another example of a prompt template that provides a single more\_infos value to be valued at call time;

Our class, in the service layer, is responsible for the templates used to produce the prompts for separate roles. Let's look at an example:

@Service public class CompletionManager {

ChatClient chatClient;

public CompletionManager(

@Value("classpath:/prompts/calculator-system.st") Resource calculatorSystem,

ChatClient.Builder builder) {

chatClient = builder

.defaultSystem(calculatorSystem)

.defaultAdvisors(List.of(new MessageChatMemoryAdvisor(chatMemory) ))

.build();

} …

}

Notes:

* The message template for the SYSTEM role is read from an external file located in the classpath because it is more convenient to access and modify by developers, prompt engineering, etc.;
* The class is marked with the service stereotype, it is placed at the same application layer as the repositories, as we said;
* ChatClient plays a similar role to JDBC drivers and is instantiated and stored in our class;

Let's see another simple template, in this case without variables:

You are a helpful and kind agent, you know about flowers and the world production of various nations.

Respond with a simple list of countries, without further comment.

Here the method to call the model:

public String chatResponse(String message) {

return chatClient.prompt()

.advisors(new SimpleLoggerAdvisor())

.advisors(new ClearThreadUnsafeLoggerAdvisor())

.user(message)

.call()

.content();

}

Note:

* The user’s request is conveyed through the string *message*;
* We are using an Advisor wrote by me to log the communication with the LLM in a clearer form. It is a deeply insecure class, with the only merit of logging the chat in a more comfortable way for the readers of this article. In production, please use SimpleLoggerAdvisor, powered by Spring AI;

Finally, here's an example of a client in our class:

completionManager.chatResponse("How much is 3 times 3?");

And related answer:

request:

SYSTEM: You are a helpful and kind agent, you know about flowers and the world production of the various nations.

Respond with a simple list of countries, without further comment.

USER: Which are the top three tulip exporting nations?

response:

ASSISTANT: 1. Netherlands

2. Poland

3. Germany

The AI’s responses, evidently, are not produced by our applications so there is no template for them: however, it is possible to instruct the model to respond to us in a particular format, for example in JSON. Until recently, it was not so easy to convince an LLM to return only JSON because it often anticipated it with an enthusiastic comment like: *“of course I can respond with a JSON here it is:”*.

Now you can force the response format out of the prompt, directly in the service call:

public Works entity(String message) {

var outputConverter = new BeanOutputConverter<>(CompletionManager.Works.class);

ChatResponse response =chatClient.prompt()

.advisors(new ClearThreadUnsafeLoggerAdvisor())

.system("You are a fine man of letters, a lover of Polish poetry and eager to make it known all over the world.")

.options(OpenAiChatOptions.builder()

.withResponseFormat(new ResponseFormat(ResponseFormat.Type.JSON\_SCHEMA, outputConverter.getJsonSchema()))

.build())

.user(message)

.call()

.chatResponse();

String content = response.getResult().getOutput().getContent(); }

return outputConverter.convert(content);

Notes:

* Using the OpenAIChatOptions class, we're passing in model-specific options. In addition to indicating the response format, there are many other options, such as the temperature of the response;
* We used a different model from OpenAI, the good *gpt-4o-mini*;
* We've replaced the default SYSTEM message, just to show how this is possible;
* In the *content variable*, the JSON string returned by the template;

Now, our class has the task of returning the LLM answers in a model-independent format to make it usable by the other application layers. In the example, if the request is to have a list of the most beautiful poems by the Polish poet Wisława Szymborska, we will have to define a special DTO and return an instance.

record Works(@JsonProperty(required = true, value = "works") Work[] works,

@JsonProperty(required = true, value = "poet\_name") String poet) {

record Work(@JsonProperty(required = true, value = "title") String title,

@JsonProperty(required = true, value = "year") Integer year) {

}

}

# Memory

After countless conversations with ChatGPT or Gemini or LLama, it may perhaps be surprising to discover that LLMs are stateless services: each call is independent of the others and there is no session. Let's check it:

completionManager.chatResponse("What are the top three tulip exporting nations?");

completionManager.chatResponse("And the fourth?");

The answer will look like:

request:

SYSTEM: You are a helpful and kind agent, you know about flowers...

USER: Which are the top three tulip exporting nations?

response:

ASSISTANT: 1. Netherlands

2. United States

3. Japan

request:

SYSTEM: You are a helpful and kind agent, you know about flowers...

USER: And the fourth?

response:

ASSISTANT: I'm sorry, but I'm not sure what you're asking. Can you give me more context or clarify your question?

To be able to hold a dialog, you must report in the most recent prompt the conversation that has taken place so far.

This injection into the prompt (be it SYSTEM or USER) is often referred to as *stuff prompting* and it is the task of our application to take care of it, using the appropriate classes of the different libraries that generally take the name of Memory.

Spring AI has a couple of very basic classes for memory management; to use them you need to add the appropriate advisor to the ChatClient instance:

public CompletionManager(@Value("classpath:/prompts/calculator-system.st") Resource calculatorSystem,

ChatClient.Builder builder, ChatMemory chatMemory) {

chatClient = builder

.defaultSystem(calculatorSystem)

.defaultAdvisors(List.of(new MessageChatMemoryAdvisor(chatMemory) ))

.build();

}

After this change, let's verify the response to the newly generated conversation:

request:

SYSTEM: You are a helpful and kind agent, you know about flowers...

USER: Which are the top three tulip exporting nations?

response:

ASSISTANT: - Netherlands

-Germany

-France

request:

SYSTEM: You are a helpful and kind agent, you know about flowers...

USER: Which are the top three tulip exporting nations?

ASSISTANT: - Netherlands

-Germany

-France

USER: And the fourth?

response:

ASSISTANT: Italy

Note:

* in the second request the previous conversation was also inserted (stuff);

The ability to persist a conversation on files or databases is delegated to classes called by langchain History: this library offers multiple persistence options while Spring AI, now, only one through the CassandraChatMemory class (guess which database it saves on? 😊 )

Adding the whole conversation to the context finds its biggest obstacle in the limitation of the prompt: each LLM accepts only a certain number of characters (tokens, more precisely) and not one more. Costs would also be an issue to consider because the fees are based on tokens, which are fractions of the words entered in the prompts. For these reasons, langchain has classes, such as ConversationSummaryMemory, which summarize the previous conversation, so that only this excerpt is injected into the prompt. Obviously, at the price of more interactions with the model.

# Functions

In carrying out its task, an LLM can resort to external functions. Be careful, he is not able to invoke them, he can simply ask our application to do so.

{

"$schema": "https://json-schema.org/draft/2020-12/schema",

"name": "orderStatusService",

"description": "Returns the status of the order",

"parameters": {

"type": "object",

"properties": {

"order\_code": {

"type": "string",

"description": "The order code, e.g. XX-427"

}

},

"required": ["order\_code"]

}

}

You must pass the json schema of the functions it can call to the model, the example above refers to a function to get the state of an order; the only parameter is the order identification code. LLM only uses descriptions, function names, and parameters to understand how to use them, which must therefore be very clear and comprehensive.

The json schema is produced starting from our source code, using annotations.

@Service

public class OrderStatusService implements Function<OrderStatusService.Request, OrderStatusService.Response> {

@JsonClassDescription("Returns Order Status")

public record Request(@JsonProperty(required = true, value = "order\_code")

@JsonPropertyDescription("The order code, e.g. XX-427") String orderCode) {

}

@JsonClassDescription("Returns Order Status")

public record Response(Status unit) {

}

public enum Status {

RICEIVED, SEND, REFIUSED, DELIVERED;

public static Status getRandom() {

Status[] stati = values();

return stati[new Random().nextInt(stati.length)];

}

}

public Response apply(Request request) {

return new Response(Status.getRandom());

}

}

Note:

* There are many ways to configure the functions to be passed to the LLM, here I present an alternative way to the official documentation, defining a component of Spring in which it is possible to inject other beans of the context;
* The descriptions passed to the LLM are borrowed from the component and json annotations, remember to add them and describe them with simplicity and clarity;
* Java is not a functional language, despite the introduction of lambdas and functional interfaces way back in Java 8. Passing a function to the LLM requires a little more code than dynamic languages. Our component, therefore, implements the default functional interface of the JDK, where the invocation takes place through the apply method;
* In this example, the order status is returned randomly among the values of the Status enumerator;

The functional must be passed to the ChatModel instance; we do it in another component, as in the previous examples:

@Service

public class AgentManager {

ChatClient chatClient;

public AgentManager(@Value("classpath:/prompts/flowers-system.st") Resource flowersSystem,

ChatClient.Builder builder) {

chatClient = builder

.defaultSystem(flowersSystem)

.defaultAdvisors(new ClearThreadUnsafeLoggerAdvisor())

.defaultOptions(OpenAiChatOptions.builder()

.withFunction("orderStatusService")

.build())

.build();

}

public String tools(String message) {

return chatClient.prompt()

.user(message)

.call()

.content();

}

}

Note:

* We find the only novelty is among the options of the model, where we pass the reference to the bean called orderStatusService;

In the dialogue between the application and the model, the TOOL role, the fourth and last type of message, intervenes.

request:

SYSTEM: You are the virtual assistant of an online bookstore. Respond to users with extreme kindness, remind them to always specify the order code, greet by remembering the name of the company: "Scripta Manent"

USER: What status is the order with code AB-621 in?

response:

ASSISTANT: I'll be happy to help you! A moment, please, while I check the status of the order with code AB-621. orderStatusService({"order\_code":"AB-621"})

… our application invokes the function

TOOL: ToolResponse[id=call\_527S70lYLcYLAgeKe4GmWjcc, name=orderStatusService, responseData={"status":"RICEIVED"}]

response:

ASSISTANT: Your order with code AB-621 is currently being shipped. Thank you for contacting us and have a nice day from Scripta Manent!

The assistant asks the application to invoke a specific function for which it passes the {"order\_code":"AB-621"} parameters. Spring acts as an agent and makes the call for us, passing the values obtained to the LLM {"status":"RICEIVED"} with the next call, using a TOOL message.

Note:

* My log class fails to report agent messages because they happen inside the *OpenAIChatModel* class, before each advisor is invoked. You can still retrieve them by debugging the *OpenAIChatModel.doGetChatResponse method*

Or raise the logging level of the HTTP client in the *application.yaml*:

logging:

level:

org.springframework.web.client: INFO

The model may require the subsequent invocation of several functions before responding to the user's prompt.

# In langchain nomenclature, a Chain capable of invoking functions is called an Agent while an AgentExecutor run one (or more) Agents until the LLM requests to invoke functions. Essentially, it handles the part of the conversation that requires functions to be performed. In Spring AI, as we have seen, it is the only ChatModel class to act as an agent in case functions are specified.

# Embeddings

A natural language sentence can be classified by several characteristics, and each characteristic can be scored between -1 and 1. For example, does the sentence talk about Paris? Does it have a positive feeling? Do sunsets have anything to do with it? Are we talking about the color red? Are we talking about computers? Are we talking about Apple computers?

At each question or, more precisely, at each quality of the text is assigned a score and the list of these scores forms an array (or a vector/tensor if you are a mathematician) called embedding, characterized by high dimensionality. ChatGPT, for example, uses 1536 scores.

Let's look at a class for producing embeddings:

@Service

public class EmbeddingsManager {

@Autowired

EmbeddingModel, embeddingModel;

public EmbeddingResponse encode(List<String> texts) {

if (isEmpty(texts)) {

texts = List.of("Hello World", "World is big and salvation is near");

}

return embeddingModel.call(

new EmbeddingRequest(texts,

OpenAiEmbeddingOptions.builder()

//.withModel("text-embedding-ada-002")

.build()));

}

}

Note:

* EmbeddingModel is the prototype of the class to produce embeding using different models, one for concrete implementation;
* The commented line shows how to select a specific model from OpenAI;

Here's a client to create a couple of vectors:

EmbeddingResponse encoded = embeddingsManager.encode(List.of("Proviamo con qualcosa"));

A screenshot of a computer

Description automatically generated

Note:

* The debug session shows the embedding created, only one (size = 1) with all 1536 floats between -1 and 1 that compose it;

An embedding only makes sense, it is understandable, only from the model that generated it, so make sure you use the same model for both creation and use.

# Vector Database

If the embedding represents the content of a text, therefore its semantics, why not persist both, text and vector, on a table? To do this, you need a vector database, also called vector store, so one that can persist and search for vectors:

CREATE TABLE IF NOT EXISTS public.vector\_store

(

id uuid NOT NULL DEFAULT uuid\_generate\_v4(),

content text COLLATE pg\_catalog." default",

metadata json,

embedding vector(1536),

CONSTRAINT vector\_store\_pkey PRIMARY KEY (id)

)

The DDL (Data Definition Language) above refers to the historic Postgresql, transformed into a vector store by the pgvector extension that threw it into the fray of the new vector databases, which have sprung up on the market like mushrooms in recent years.

The columns are easy to understand:

* **Id**: A uuid used as the primary key;
* **Text**: the plain text;
* **Metadata**: metadata in json format;
* **Embedding**: our vector calculated by LLM;

Spring AI, and certainly langchain, allow you to save texts, metadata and embeddings in a very simple way:

List<Document> documents = List.of(

new Document(getText("classpath:/data/fatti\_curiosi.txt"), Map.of("meta1", "meta1"))

);

vectorStore.add(documents);

Let's check that the data are entered in the table:

A screenshot of a computer

Description automatically generated

Business documents are stored in a wide variety of formats (pdf, word, json, etc.), so specific classes are needed to read them. Langchain and the llamaindex python library allow you to work with dozens of different types. Spring AI also offers features for the most common formats, such as word:

List<Document> readWordDoc() {

var wordDoc = new DefaultResourceLoader().getResource("classpath:/etl/article.docx");

TikaDocumentReader tikaDocumentReader = new TikaDocumentReader(wordDoc);

TokenTextSplitter splitter = new TokenTextSplitter(500, 400, 10, 1000, true);

return splitter.apply(tikaDocumentReader.read());

}

Note:

* For this kind of document, Spring AI delegates reading to the Apache Foundation's excellent Tika library;
* The code loads the Microsoft word version of the article you are reading;
* The last few lines deal with breaking the entire document into smaller parts, sometimes called chunks;

The latter is a necessary operation because LLMs have insurmountable limits relating to the length of the prompts and the whole document may not fit in them. Also, the larger the document to be analyzed, the longer it takes for the model to respond. Finally, by putting elements unrelated to the user's request into context, it could confuse the LLM and lead it to respond incorrectly.



We then divide the document into smaller parts: in the example we are setting the default, minimum and maximum sizes (in tokens and characters) and putting the ‘\n’ in the chunks.

A sometimes better approach is to divide by paragraphs, but much depends on the document dealt with so, unfortunately, there is no general rule for this operation so relevant to the quality of the LLM responses.

# Retrieval

Now, any data is saved in a db just to be able to search and extract it quickly. The novelty offered by vector databases consists in being able to carry out semantic searches, i.e. find documents whose content is similar to a past one. Similar content means nearby embedding vectors.

The steps are quite simple:

public Collection<Document> retreive(String question) {

var fb = new FilterExpressionBuilder();

return vectorStore.similaritySearch((

SearchRequest.query(question)

.withTopK(3)

.withFilterExpression( fb.eq("source", "article.docx").build())

));

}

Notes:

* The embedding is calculated for the text passed by the user, the one of which we have to find similar documents in our vector store. I repeat here the warning to use the same model for both saving and searching;
* Leave it to the vector database to extract the first *k* most similar documents, exactly three in the example
* We set a search filter on metadata;

A diagram of a line and a point

Description automatically generated with medium confidence

In the search for embeddings close to the one passed by the user, vector databases generally use two types of algorithms:

* L2 squared: it is nothing more than the square of the (Euclidean) distance between two points (vectors) in the (hyper-)plane, the green line in the graph on the left;
* Cosine of similarity: given the two points (vectors), the two semi-straight lines are drawn towards the origin of the axes. The cosine of the angle of these two lines is often the most widely used metric for the distance between vectors;

# Retrieval augmented generation

Each LLM has been trained on millions of datasets, therefore, he knows many topics in the public domain but not all of them and certainly knows nothing about our documents, those internal to companies, procedures, products.

To enable him to answer these issues as well, one could proceed with training but, since it is an expensive task in terms of resources and time, therefore of money, a shorter way is often used: RAG applications.

The operation, in general, is easy to understand:

* the user asks for something (e.g. "How can I ask for a refund?");
* (Retrieval) we perform the same search on our vector databases and extract the most relevant documents, for example, three chunks from the pdf relating to the internal refund procedure;
* (Augmentation) insert in the message with the role SYSTEM the contents retrieved in the previous point...
* (Generation) ... so that the LLM can use them to respond to you;

Here's a sample chat using this very article to increase the context of the LLM:

request:

SYSTEM: You are the assistant of a development team. Your job is to stay informed about new technologies,

Read scientific articles and answer questions from developers.

USER: Is there an analogy between legacy db app and LLL app?

Context information is below.

---------------------

RAG apps for legacy spring developers

Introduction to RAG apps and Spring AI

With a continuous confrontation his langchain.

A SQL analogy

Our legacy applications interface with those peculiar external services that are relational dbs using simple text structure in the SQL language. Interaction with LLMs is also configured as a call to services intermediated by simple text, in this case called prompt, structured but in natural language, ours.

(omissis...)

---------------------

Given the context and provided history information and not prior knowledge,

reply to the user comment. If the answer is not in the context, inform

the user that you can't answer the question.

response:

ASSISTANT: The analogy between legacy database applications and large language model (LLM)-based applications is that both use structured text to interact with external services. In the case of legacy applications, structured text is SQL, while for LLM applications, structured text is a natural language prompt.

Both types of applications need some sort of abstraction layer to handle communication with external services. For legacy applications, this is often handled by a DAO class and the use of database-specific drivers (such as ODBC or JDBC). For LLM applications, libraries such as Langchain provide the abstraction needed to interact with different language models.

If you have any further questions or need more details on a specific aspect, please let me know!

Note:

* the documents retrieved from the vector database have been added to the USER message and the LLM has been instructed on how to use them (from the phrase Context information is below);
* The part of the template used by the RAG is in English: ChatGPT understands the instructions and does not confuse the two languages by answering in Italian, that of the user. Spring Ai allows, in any case, to easily replace the template in the constructor of the QuestionAnswerAdvisor class;

RAG features are added to ChatClient through the QuestionAnswerAdvisor in Spring AI:

public RAGManager(@Value("classpath:/prompts/rag-system.st") Resource flowersSystem,

ChatClient.Builder builder, ChatMemory chatMemory, OpenAiChatModel openAiChatModel, VectorStore vectorStore) {

chatClient = builder

.defaultSystem(flowersSystem)

.defaultAdvisors(List.of(new MessageChatMemoryAdvisor(chatMemory),

new QuestionAnswerAdvisor(vectorStore, SearchRequest.defaults()

.withTopK(3)),

new ClearThreadUnsafeLoggerAdvisor())

.build();

}

The call is very similar to the previous ones:

public String ragResponse(String message) {

return chatClient.prompt()

.user(message)

.advisors(a -> a.param(QuestionAnswerAdvisor.FILTER\_EXPRESSION, "source == 'article.docx'"))

.call()

.content();

}

Note:

* Let's see here a different way to set a filter on database metadata;

All simple, perhaps too simple; the current implementation of Spring AI is still extremely basic: using the last user prompt for semantic search can be very misleading, as evidenced by the following chat.

request:

USER: There is an analogy between legacy db app and LLL app

response:

ASSISTANT: Of course, blah blah blah

request:

USER: tell me more?

Searching on the vector store for something similar to "tell me more" will hardly lead to the desired result. The Chain designed for langchain's RAG is called ConversationalRetrievalChain and solves the problem by using, within it, two different chains: CondenseQuestionChain takes care of summarizing the conversation and using the summary, including the last prompt, for semantic search; the second, CombineDocsChain uses the output of the first to respond to the user.

In fact, the whole theme offers ample room for improvement for Spring AI: the *k* documents retrieved from the database can, of course, be entered directly into the prompt (*prompt stuff*), or they could be evaluated with a score to include only the best ones (*map\_rerank*), or synthesized, as mentioned, or even refine the search from one document to another (*refine*). These and other possibilities are offered by langchain only.

# Conclusions

Spring AI seems like a very well-engineered library as, on the other hand, is the tradition of the entire Spring Framework. At present, the foundations have been laid for future developments that we expect to fill the considerable gap in functionality compared to langchain, trusting to be able to write our next RAG application in Java.

# In this regard, I would like to point out the existence of the langchain4j library, a project independent from langchain but openly inspired from it: it would certainly be interesting to compare its architecture with that of Spring AI, perhaps in a future article.

# Bibliography

<https://docs.spring.io/spring-ai/reference> the official documentation of the new Spring AI library

<https://platform.openai.com/playground> a web console where you can interactively try out prompts, tools, and calls to OpenAI models

<https://python.langchain.com> the official documentation of the world-famous Python Langchain library

<https://llamaindex.ai> a python library to persist documents in various formats and in a multitude of different vector databases

<https://github.com/ggerganov/llama.cpp> an open-source project to deliver different models, even locally

<https://github.com/vllm-project/vllm> another project similar to the previous one to provide several LLMs also locally

<https://www.h2database.com> an excellent in-memory database

<https://www.postgresql.org> a formidable database, as long-lived as the elephant that symbolizes it

<https://github.com/pgvector/pgvector> Postgresql extension to persist and search for embedding vectors

<https://github.com/langchain4j/langchain4j> independent project compared to langchain but with a similar architecture