- Spring AI is a Spring Boot abstraction layer that lets your Java app talk to AI models (LLMs) like OpenAI, Azure OpenAI, Ollama, HuggingFace, etc.
  - You do NOT write raw OpenAI REST calls.
  - You do NOT manage prompts manually everywhere.
  - You do NOT bind your code to one AI vendor.
- **Spring AI doesn't create models; It connects to existing models;**
- **Dependencies of Spring AI**
  - **spring-ai-bom** (version control)
    - Keeps all Spring AI modules on compatible versions
    - Prevents dependency conflicts
    - Nothing functional; only dependency management.
    - Inside this you can see something like this

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.ai</groupId>
      <artifactId>spring-ai-commons</artifactId>
      <version>${project.version}</version>
    </dependency>
    <dependency>
      <groupId>org.springframework.ai</groupId>
      <artifactId>spring-ai-template-st</artifactId>
      <version>${project.version}</version>
    </dependency>
```

  - **spring-ai-client-chat** (chat API abstraction)
    - Gives you **ChatClient**
    - Unified way to *send prompts & get responses*

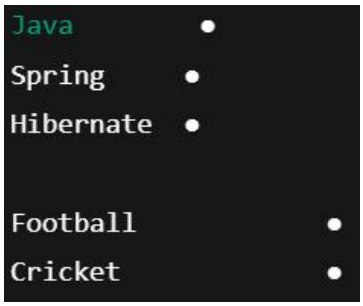
      without this, no chat with LLMs.
  - **spring-ai-model** (model contract layer)
    - Defines interfaces like **ChatModel**, **EmbeddingModel**
    - Makes providers interchangeable

      This is the brain of Spring AI.
  - **spring-ai-starter-model-openai** (OpenAI integration)
    - Auto-configures OpenAI client.
    - Reads **API key, model, temperature** from properties

      without this, Spring AI doesn't know OpenAI exists.
  -

ChatModel vs ChatClient vs EmbeddingModel

- OpenAI provides Chat models and Embedding models

  these both does separate things.
  - You can use: chat only, embedding only *or* both (RAG).
  - ChatClient/ChatModel handle **text generation**, EmbeddingModel handles **vectorization**; they are architecturally and functionally independent in Spring AI.
  - Vectorization is required to perform semantic search over large text corpora; ChatClient only generates text and cannot retrieve or rank information, so embeddings convert language into a numeric space where similarity can be computed.
    - For example think over a large pdf and you asked some question related to the context at some page.
    - here, vectorization is used to find the most related topic of your search.
    - All keywords are represented with a number, and numbers having smallest distance between then are chosen to be related.
    - Now after finding the related content, chat client is used for text generation.
    - 

- **ChatModel**
  - Lowest level (actual LLM wrapper)
  - What it is?
    - Interface over the real LLM
    - One implementation per provider (OpenAI, Ollama, Claude …)
  - What is does?
    - Sends request to model
    - Gets raw response
    - No memory, no prompt templates, no tools
  - Analogy: this is Engine.

- Example

```
ChatModel chatModel; // OpenAiChatModel, OllamaChatModel, etc
```

- **ChatClient**
  - Developer-friendly facade over ChatModel
  - What it is?
    - High-level API built on top of ChatModel
    - This is what YOU should use in apps
  - What it adds?
    - Prompt building
    - System / user messages
    - Advisors (memory, RAG, tools)
    - Fluent API
  - Analogy: This is the steering wheel + dashboard.
  - Example

```
ChatClient chatClient;

chatClient.prompt()
    .user("Explain JVM")
    .call()
    .content();
```

- **EmbeddingModel**
  - Text → Vector converter
  - What it is?
    - Separate model type
    - NOT for chatting
  - What it does?
    - Converts text into numerical vectors
    - Used for RAG / semantic search
  - Analogy: This is for search, not talking.
- Example
  -

```
EmbeddingModel embeddingModel;

float[] vector = embeddingModel.embed("Spring AI explained");
```

Example of working of both Chat client and Embedding model

- The problem statement is: **Ask questions and get answers ONLY from those docs** (PDF, doc or any other)
- **Step 1: Convert documents to vectors**

```
@Autowired
EmbeddingModel embeddingModel;

@Autowired
VectorStore vectorStore;

public void indexDocs() {

    String doc1 = "Spring Boot supports dependency injection using @Autowired";
    String doc2 = "JWT is commonly used to secure REST APIs";

    vectorStore.add(List.of(
        new Document(doc1),
        new Document(doc2)
    ));
}
```

  - Behind the scenes:

    Text → Embedding model → numbers → stored in Vector DB

    (this happens once; indexing phase)

- **Step 2: User asks a question**

  - `"How do I secure a Spring Boot API?"`

- **Step 3: Convert question to vector**

  - Behind the scenes

    Question → Embedding model → vector

- **Step 4: Vector search (meaning-based)**

  -
```
Vector DB finds:
"JWT is commonly used to secure REST APIs"
```
  - Keywords match: NO ❌
  - Meaning match: YES ✅

    this is the exact thing vector does

- **Step 5: Generate answer (ChatClient)**

  - Now we finally uses ChatClient

```
@Autowired
ChatClient chatClient;

public String ask(String question) {

    return chatClient.prompt()
        .system("""
            Answer ONLY using the provided context.
        """)
        .user(question)
        .call()
        .content();
}
```

- What actual prompt sent to LLM?

```
Context:
JWT is commonly used to secure REST APIs

Question:
How do I secure a Spring Boot API?
```

- LLM generates

```
"You can secure a Spring Boot API using JWT-based authentication..."
```

➢ **Final mental model**

- **EmbeddingModel : finds What to read**
- **ChatClient : explains What was found**
- Embeddings are used to retrieve relevant context via semantic search, and ChatClient uses that context to generate a final natural-language answer.

- **Temperature**
  - It controls the randomness in text generation.
  - Low temperature → safe, predictable choice
  - High temperature → risky, creative choice
  - Prompt

    ```
    "Spring Boot is used for"
    ```
  - Temperature: 0.0

    ```
    Spring Boot is used for building Java-based web applications.
    ```
  - Temperature: 0.5

    ```
    Spring Boot is used for creating production-ready backend services.
    ```
  - Temperature: 1.0

    ```
    Spring Boot is used for rapidly assembling modern server-side systems with minimal configuration.
    ```
  - Temperature: 1.5+

    ```
    Spring Boot is used for powering scalable digital ecosystems across enterprises.
    ```
Use case	Temperature
Factual answers	`0.0 – 0.3`
Coding	`0.0 – 0.2`
APIs / docs	`0.1 – 0.3`
Chatbot	`0.5 – 0.7`
Brainstorming	`0.8 – 1.2`
Story / creativity	`1.0+`
- **Top P**
  - Decrease the number of possibilities.
  - If **Top P = x**, 0 <= x <= 1
    - It'll add the probabilities of the possible output (from high to low), and when the sum of probabilities reach **x**, it stops.
    - Only those many outputs are chosen out of which one will be randomly selected depending upon the ***temperature*** value.

- Top-P limits token selection to the smallest set whose cumulative probability exceeds P, controlling output diversity.
- Top-P = 0.1
  - Only most likely word(s)
  - Very deterministic
- Top-P = 0.9
  - Many reasonable options
  - Balanced output
- Top-P = 1.0
  - No restriction (because sum of all words's probabilities will be 1)
  - Full vocabulary

➢ **Top K**

- ***Top P*** limits the number of probable outputs according to the sum of probabilities.
- ***Top K*** limits the number of probable outputs by its count.
- If you give

  Top K = 5
  - Then only 5 probable outputs will be selected, out of which an output is randomly choosen according to ***temperature***.
- Either you can choose ***Top P*** or ***Top K***

➢

- We'll use Ollama here, download it and pull any AI model using the command

  ollama pull <AI model name>

  - ```
    ollama pull qwen3:0.6b
    ```
  - If you want to see all the models you have pulled, use **ollama list** command
    - ```
      $ ollama list
      NAME          ID              SIZE      MODIFIED
      qwen3:0.6b    7df6b6e09427    522 MB    7 seconds ago
      ```
  - To run the model, use **ollama run** <AI model name>
    - ```
      ollama run qwen3:0.6b
      ```
    - use **--verbose** to get more details

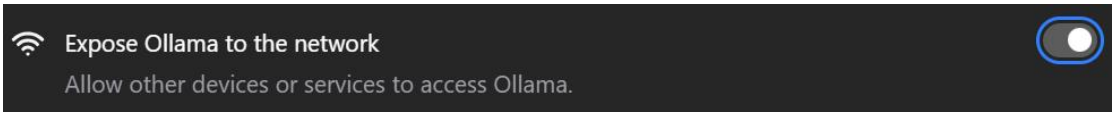
      ```
      ollama run qwen3:0.6b --verbose
      ```

- You need to configure **application.properties** or **application.yml** file
  - ```yaml
    spring:
      ai:
        openai:
          api-key: ${OPENAI_API_KEY}

          chat:
            options:
              model: gpt-4
              temperature: 0.2
              max-tokens: 500
              top-p: 1.0
              frequency-penalty: 0.0
              presence-penalty: 0.0
    ```
    (example)

    openai is the AI model provider, **chat.model** is the actual model to be used.

- For **Ollama**
  - Open *Ollama* ***settings*** and enable the "Expose Ollama to the network" option.

  - It is by default exposed on port ***11434*** on localhost.
- For **OpenAI**
  - Create one secret key in platform.openai.com/settings/organization/api-keys

- For ***ollama***, you need to give the URL, for ***openai*** you need to give the **api-key**.
  - Remaining options you can give as you wish.

```yaml
spring:
  application:
    name: spring-ai

  ai:
    ollama:
      base-url: http://localhost:11434
      chat:
        options:
          model: qwen3:0.6b

    openai:
      api-key: sk-proj-lNttdv2eFnmgj_SC4V
      chat:
        options:
          model: gpt-4o-mini
          temperature: 0.9
```

  - You need to create a bean of the ***ChatClient***

```java
@Bean
public ChatClient chatClient( @NotNull ChatClient.Builder builder) {
    return builder.build();
}
```

    - Depending upon the dependency, it'll create the client.
    - If the ollama dependency is there, then chat client will be of Ollama;
    - if openai dependency is there, then chat client will be of openai.

```java
public String getJoke(String topic) {  1 usage
    return chatClient.prompt()  ChatClientRequestSpec
            .system( s: "You are a sarcastic joker, give response in 1 line only.")
            .user( s: "Give me a joke on the topic: " + topic)
            .call()  CallResponseSpec
            .content();
}
```

  - you can talk to LLM like this, **system** means system prompt, **user** means user prompt.

```
public String getJoke(String topic) {  1 usage

    String systemPrompt = """
            You are a sarcastic joker, you make poetic jokes in 4 lines.
            You don't make jokes about politics.
            Give a joke on the topic: {curr_topic}
            """;
    PromptTemplate promptTemplate = new PromptTemplate( template: systemPrompt);
    String renderedText = promptTemplate.render( additionalVariables: Map.of( k1: "curr_topic",  v1: topic));

    return chatClient.prompt() ChatClientRequestSpec
            .user( s: renderedText)
            .call() CallResponseSpec
            .content();
}
```

you can also use like this.

Note: key of the map (***curr_topic*** in my case) should be same as that inside the curly braces inside that string (in my case **{*curr_topic*}**).

- if you see in debugger mode, you'll be able to see the *renderedText*

```
You are a sarcastic joker, you make poetic jokes in 4 lines.
You don't make jokes about politics.
Give a joke on the topic: programmers
```

- F
- F
- F
- F
- F
- F
- F
- F
-

➢ ChatModel, ChatClient in Spring AI

- In Spring AI, the interface **ChatModel** present, which talks to LLM (http request)
- For different providers like ***ollama, openai, claude*** ..etc, Spring AI contains concrete class implementing **ChatModel** .

  You need to add the dependency for specific providers like *spring-ai-starter-model-openai*, *spring-ai-starter-model-ollama* ..etc.
- **ChatClient** is a interface, which is implemented by the concrete class **DefaultChatClient**.
  - ChatClient contains the object of **ChatModel**.
  - Whatever the providers are, the **ChatClient** and **DefaultChatClient** are constant.
  - The only thing that ***varies*** is the *concrete implementation of ChatModel interface* for different providers.
  - ```
    public class OpenAiChatModel implements ChatModel {
    ```

    it is the **openai** chat model
- **ChatClient** just gives a good abstraction handling all the stuffs and gives you a good organized response.

```
Your Code
   ↓
ChatClient   ← façade you program against
   ↓
ChatModel    ← provider-specific model adapter
   ↓
LLM API (HTTP)
```
```

➢ ChatClient.Builder Auto-configuration

- Inside the auto-configuration class org.springframework.ai.model.chat.client.autoconfigure, the **ChatClient.Builder()** bean is created.
- If there is only one provider's dependency i.e. ***openai, claude, ollama*** or something else, then it creates a bean of ***ChatClient.Builder*** class.
  But in case of multiple provider's dependencies, it doesn't create the bean.
- But however, you need to create a **ChatClient** bean using the ChatClient.Builder bean in case of single provider's dependency.

```
@Configuration
public class AIConfig {

    @Bean
    public ChatClient chatClient( @NotNull ChatClient.Builder builder) {
        return builder.build();
    }

}
```

Here you are getting the bean of **ChatClient.Builder** because Spring AI has auto-configured this (I have given only **openai** dependency in *pom.xml*)
if I had multiple provider's dependency, then I won't have this ChatClient.Builder bean.

- There is something like this

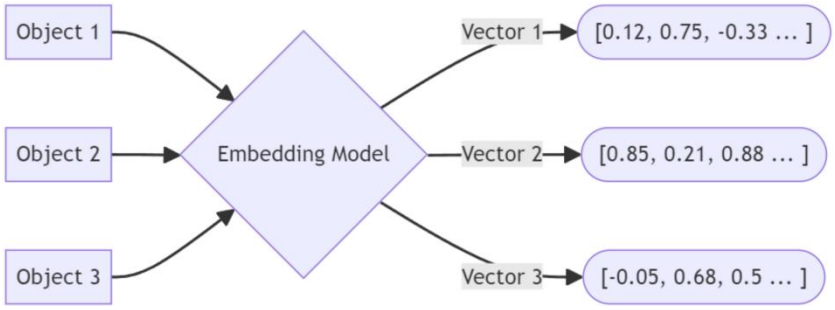
```
@AutoConfiguration
@ConditionalOnSingleCandidate(ChatModel.class)
class ChatClientAutoConfiguration {

    @Bean
    ChatClient.Builder chatClientBuilder(ChatModel chatModel) {
        return ChatClient.builder(chatModel);
    }
}
```

here, if multiple AI providers are there, then this **@ConditionalOnSingleCandidate(ChatModel.class)** will fail resulting in not creating bean of ChatClient.Builder

Embedding & Vector search

- **Embedding**
  - It is basically **semantic compression** into a **fixed-size numeric vector.**

    The size is fixed per model and never changes per input.
  - Analogy: in color picker, RGB values are stored like for green (0,255,0), like this, the vector is represented for each text.
  - Length of this vector is called dimensions.
    - In vector (physics) we store using the dimensions only (usually 3)
  - **An embedding model always outputs vectors of the same dimensionality for every input.**
    - The dimensionality is **decided by model**, not by input length, word length, or sentence complexity.
  - 

  - ```
    "Java"        → [0.12, -0.98, 0.44, ...]  ← length = N
    "Spring Boot" → [0.09, -1.02, 0.41, ...]  ← length = N
    "AI"          → [0.15, -0.91, 0.47, ...]  ← length = N
    ```
Model family	Vector size
Small models	384
Medium models	512 / 768
Large models	1024 / 1536 / 3072
Feature	ChatModel	EmbeddingModel
Output	Variable-length text	Fixed-length vector
Use case	Generation	Similarity search
Shape	Unbounded	Strictly fixed
Stored in DB	❌	✅
    ```

- **Embedding Dimensions**
  - The vector is of fixed length. Length is called Embedding dimension.
  - Each dimension is a learned latent feature.

    Latent: hidden, not human-readable
    - Each dimension is a number slot.
    - The model has learned how to use each dimension during training.
  - The model learns how to distribute meaning across all dimensions.
    - Single dimension has no meaning;
    - meaning exist only in combined pattern of each dimension.
    - If you change one dimension, meaning barely changes;

      If you change multiple dimension, meaning changes a lot.
  - Meaning comes from the relative geometry, not individual values.
    - Its related to previous point.
    - The model compares 2 vectors depending upon the direction, angle, distance ..etc.

      If they are close, then it'll consider this.
  - **Embeddings do not store meaning in values; they store meaning in relationships between vectors.**
  - **Semantic capacity**: ability to search by meaning
  - Don't get confused between *Embedding Dimensions* and *Model Parameters*
    - Embedding Dimension : Size of output vector
      - `Embedding dimension = 768`
      - `"text" → [v1, v2, v3, ..., v768]`
    - Model Parameters: internal weights of neural network
      - `110M parameters`
        `7B parameters`
        `70B parameters`

Embedding model in Spring AI

- Interfaces & Abstraction
  - In Spring AI, there is an interface **EmbeddingModel** and an abstract class **AbstractEmbeddingModel** (it implements *EmbeddingModel*).
  - There are concrete classes extending this *AbstractEmbeddingModel* for different providers.

```
EmbeddingModel              ← interface (fixed contract)
    ↑
AbstractEmbeddingModel      ← abstract base (shared logic)
    ↑
-------------------------------------------
| OpenAiEmbeddingModel      ← OpenAI
| OllamaEmbeddingModel      ← Ollama
| AzureOpenAiEmbeddingModel ← Azure
| BedrockEmbeddingModel     ← AWS
| etc.
```

- You need to update the **application.properties** or **application.yml** file for embedding

```yaml
openai:
  api-key: sk-proj-lNttdv2eFnmgj_SC4
  chat:
    options:
      model: gpt-4o-mini
      temperature: 0.9
  embedding:
    options:
      model: text-embedding-3-small
```

- Basic usage:

```java
private final EmbeddingModel embeddingModel;


public float[] getEmbedding(String text) {
    return embeddingModel.embed(text);
}
```

- There are database to store this vectors, which is called vector database.

  These databases are not normal databases.

- To store the vectors in database, abstraction is there in Spring AI
  - **VectorStore** is the interface, which is implemented by the abstract class **AbstractVectorStore**.
    - These 2 are ***constant*** throughout.
  - Now, concrete classes extending this **AbstractVectorStore** are there depending upon each vector database.

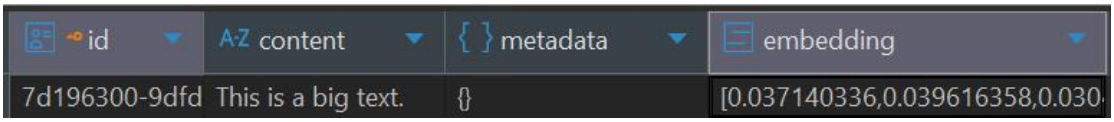
```
VectorStore (interface)          ← fixed contract
    ↑
AbstractVectorStore              ← shared base (optional)
    ↑
-------------------------------------------------
| PgVectorStore                  ← PostgreSQL + pgvector
| PineconeVectorStore             ← Pinecone
| MilvusVectorStore               ← Milvus
| RedisVectorStore                ← Redis
| ChromaVectorStore               ← Chroma
| OpenSearchVectorStore           ← OpenSearch
```

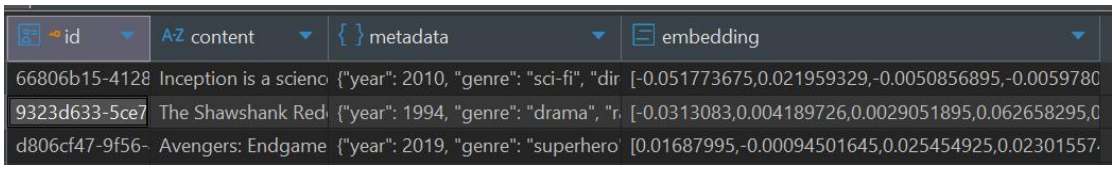
- **Configuring PG vector (postgres)**
  - Add this dependency

```
<dependency>
    <groupId>org.springframework.ai</groupId>
    <artifactId>spring-ai-starter-model-openai</artifactId>
</dependency>
```

  - Configure **application**

```
spring:
  application: {1 key}
  ai:
    ollama: {2 keys}
    openai: {3 keys}
    vectorstore:
      pgvector:
        initialize-schema: true
  datasource:
    url: jdbc:postgresql://localhost:5440/my-pg-vector-db
    username: my-user
    password: password
    driver-class-name: org.postgresql.Driver
```

- There is one class **Document** by Spring AI
  - Document represents one piece of content (text) + its metadata, **ready to be embedded, stored, and retrieved from a vector store.**
  - ```
    public void ingestDataToVectorStore(String text) {
        Document document = new Document( content: text);
        vectorStore.add(List.of( e1: document));
    }
    ```
  - 

id	content	metadata	embedding
7d196300-9dfd	This is a big text.	{}	[0.037140336,0.039616358,0.030
    - The embedding column will contain the dimensions.
- Adding and Retrieving data
  - I stored the following list of Document object in vectorStore
    - ```
      List<Document> movieDocuments = List.of(
              new Document(
                      text: "Inception is a science fiction movie directed by Christopher Nolan that explores dreams within dreams.
                      metadata: Map.of( k1: "genre", v1: "sci-fi", k2: "director", v2: "Christopher Nolan", k3: "year", v3: 2010)
              ),
              new Document(
                      text: "The Shawshank Redemption is a drama film about hope and friendship set inside a prison.",
                      metadata: Map.of( k1: "genre", v1: "drama", k2: "year", v2: 1994, k3: "rating", v3: "IMDB Top")
              ),
              new Document(
                      text: "Avengers: Endgame is a Marvel superhero movie that concludes the Infinity Saga.",
                      metadata: Map.of( k1: "genre", v1: "superhero", k2: "studio", v2: "Marvel", k3: "year", v3: 2019)
              )
      );
      ```
    - 

id	content	metadata	embedding
66806b15-4128	Inception is a scienc	{"year": 2010, "genre": "sci-fi", "dir	[-0.051773675,0.021959329,-0.0050856895,-0.0059780
9323d633-5ce7	The Shawshank Red	{"year": 1994, "genre": "drama", "r	[-0.0313083,0.004189726,0.0029051895,0.062658295,0
d806cf47-9f56-	Avengers: Endgame	{"year": 2019, "genre": "superhero	[0.01687995,-0.00094501645,0.025454925,0.02301557

      Now you can see the content (text), metadata (Map.of(…)), embeddings are present in the table
  - To perform semantic search, there is a method **similaritySearch** which is inside the **VectorStoreRetriever** interface.

    This interface is implemented by **VectorStore**

    ```
    interface VectorStore extends DocumentWriter, VectorStoreRetriever
    ```
    - It contains the **similaritySearch** method
      - ```
        public interface VectorStoreRetriever {
            List<Document> similaritySearch(SearchRequest var1);

            default List<Document> similaritySearch(String query) {
                return this.similaritySearch( var1: SearchRequest.bui
            }
        ```
        ```

```
public List<Document> similaritySearch(String text) {
      return vectorStore.similaritySearch(text);
    return vectorStore.similaritySearch(
            searchRequest: SearchRequest.builder()
                    .query( query: text)
                    .topK( topK: 2)
                    .build()
    );
}
```

ء

- F

RAG (Retrieval Augmented Generation)

- Lets say you are talking to the AI agent, and in that chat it knows about the previous conversations that you had earlier.
  - Or lets say you gave an AI agent (lets say ChatGPT) a pdf, and asking some questions related to that. How does it finds?
  - One solution is, along with user's prompt, give all the content of that PDF as well, now AI model can check those contents and give some result.

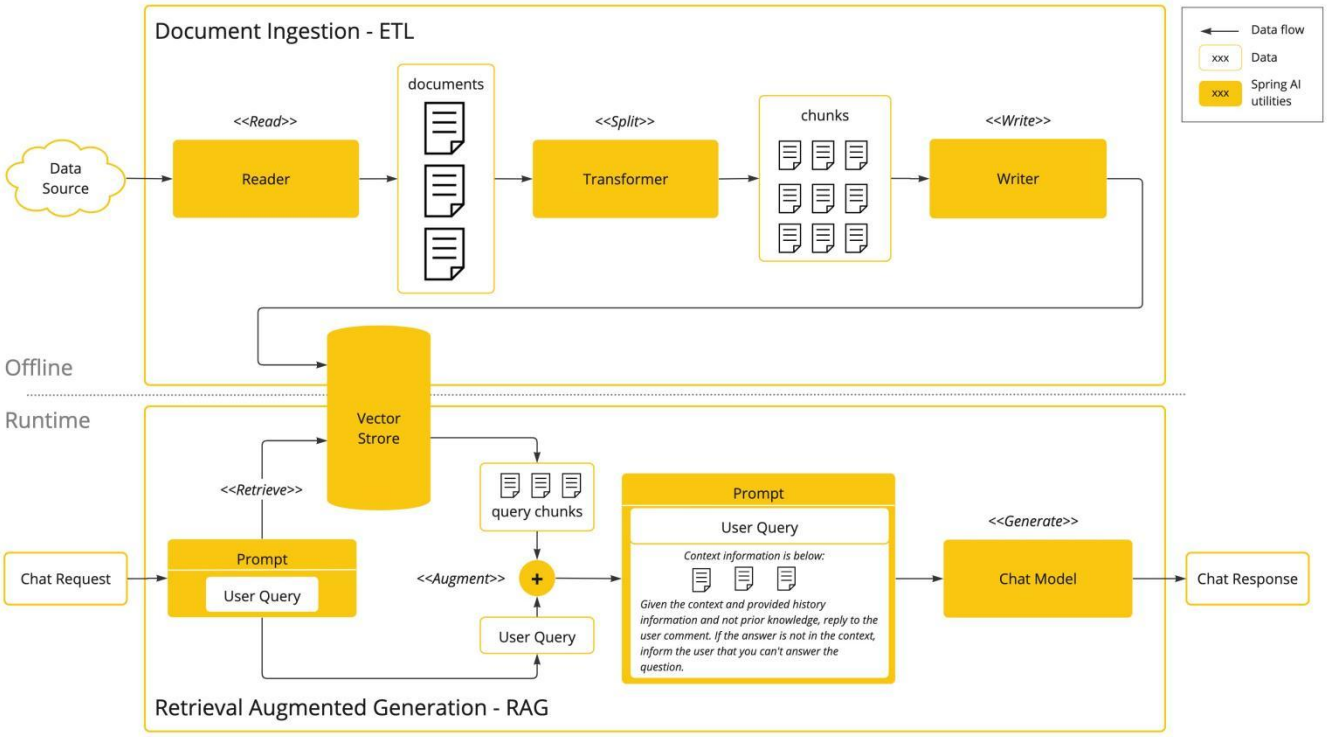
    Because LLMs alone hallucinate, & don't know your private data.

    But, with this approach, the tokens utilization will be very high, so it is not cost efficient.
  - Another solution is, find some relevant contents of that PDF according to user's prompt, and attach only those content with the user's prompt.

    Now it'll be much cheaper and optimized.
  - **RAG** = *Retrieve relevant data + Inject it into the LLM prompt before generation.*

    RAG is a concept / pattern, not a concrete entity, class, or component.
    - Retrieval : VectorStore
    - Knowledge unit : Document
    - Vectorization : EmbeddingModel
    - Injection : Advisor
    - Generation : ChatClient

➢ RAG "ETL" Pipeline (Extract, Transform, Load)



- **Phase 1 : Offline Phase : Document Ingestion (ETL)**
  - Data sources can be anything like pdf, doc, db ..etc.
  - Reader will read all those documents and then transform those into **chunks**.
    - For example, 10 pdfs are there, they are splitted into ***500*** chunks.
    - Because if you embed a big document, data can be loosed as there is fixed dimension list.
  - Then convert to embed and store this in the vector store.
  -

```
Data Source
   ↓
Reader          → reads raw files (PDF, DB, web, etc.)
   ↓
Documents
   ↓
Transformer     → splits documents into small chunks
   ↓
Chunks
   ↓
Writer          → embeds + stores chunks
   ↓
Vector Store
```

- **Phase 2 : Runtime Phase : RAG (Retrieval + Augmentation)**
  - User sends a Chat request. (user prompt)

- Attach system level prompt to that. (system prompt)
- Then retrieve the **relevant chunks** from the vector store.
- Now, along with the prompt, attach these embeddings as well to LLM.
- Now Chat model will generate and response will be sent.
-

```
User Query
   ↓
Prompt
   ↓
Vector Store (retrieve similar chunks)
   ↓
Query Chunks
   ↓
Augment (query + chunks)
   ↓
Final Prompt
   ↓
Chat Model
   ↓
Answer
```

➢ See the below experiment

- I just added a normal method to generate output based on the user's prompt only.

  Not attaching any other things like system prompt or something else.

```
public String askAI(String text) {
    return chatClient.prompt() ChatC
            .user( s: text)
            .call() CallResponseSpec
            .content();
}
```

- I told this my name

```
String res = service.askAI( text: "My name is Alok.");
```

```
Nice to meet you, Alok! How can I assist you today?
```
(output)

- Then I asked it my name

```
String res = service.askAI( text: "What is my name?");
```

```
I'm sorry, but I don't have access to personal information about users unless it's shared with me in the conversation. How can I assist you today?
```

(it doesn't know)

➢ Now we want something that knows about us, or lets say you are building company specific agent, then it should only know about you.

- It should not be a general purposed AI agent, it should be specialized for your company.
- I created a dummy list of Document object

```
public static @NotNull @Unmodifiable List<Document> springAiDocs() {  no usages
    List<Document> springAiDocs = List.of(
            new Document(
                    text: "Spring AI provides a unified abstraction for interacting with large langu
                    metadata: Map.of( k1: "source",  v1: "spring-ai-docs",  k2: "section",  v2: "overview
            ),
            new Document(
                    text: "Spring AI supports multiple AI providers such as OpenAI, Ollama, Azure Op
                    metadata: Map.of( k1: "source",  v1: "spring-ai-docs",  k2: "section",  v2: "provider
            ),
            new Document(
                    text: "Retrieval Augmented Generation (RAG) in Spring AI is implemented by combi
                    metadata: Map.of( k1: "source",  v1: "spring-ai-docs",  k2: "section",  v2: "rag")
            ),
            new Document(
                    text: "Spring AI uses Document objects as the standard unit of knowledge, contai
                    metadata: Map.of( k1: "source",  v1: "spring-ai-docs",  k2: "section",  v2: "document
            )
    );
    return springAiDocs;
}
```

- Now storing these documents in VectorStore

```
vectorStore.add(springAiDocs());
```

It automatically convert to embeddings and store in database

- As the first phase (storing the embeddings) is done, now we'll check for the retrieval and response generation part.

  First create a proper prompt to tell the AI agent to use only our data, not its own data & create a **PromptTemplate** object out of that.

```
String template = """
        You are an AI assistant helping a developer

        Rules:
        - Use ONLY the information provided in the context.
        - You MAY rephrase, summarize, and explain in natural language.
        - Do NOT introduce new concepts or facts.
        - If multiple context sections are relevant, combine them into a single explanation.
        - If the answer is not present, say "I don't know".

        Context:
        {context}

        Answer in a friendly, conversational tone.
        """;
PromptTemplate promptTemplate = new PromptTemplate(template);
```

- Now we need to give the documents as well (after fetching from the database).
  I just appended the documents (from the List<Document>) and joined those as a String.

```
List<Document> documents = vectorStore.similaritySearch(
        searchRequest: SearchRequest.builder()
                .query( query: prompt)
                .topK( topK: 2)
                .filterExpression( textExpression: "section == 'overview' or section == 'providers'")
                .build()
);

String context = documents.stream() Stream<Document>
        .map( mapper: Document::getText) Stream<String>
        .collect( collector: Collectors.joining( delimiter: "\n\n"));
```

  in that **filterExpression** block you can give the matches, these *section* is there in the metadata of the Documents list (you can check).
- Now our **PromptTemplate** and **context** is ready, we just need to create a system prompt combining those.

```
String systemPrompt = promptTemplate.render( additionalVariables: Map.of( k1: "context",  v1: context));
return chatClient.prompt() ChatClientRequestSpec
        .system( s: systemPrompt)
        .user( s: prompt)
        .call() CallResponseSpec
        .content();
```

  I just replaced that placeholder **{context}** from the template with the string version of documents list.
  both **user** and **system** prompts are given.
- I asked it now
  - "What is Apple?" ==== `I don't know.`
    because it is not stored in vector storage, and in the rule I said if the context is not there then simply say "I don't know".
- You can add advisor to do something in between. I added this to get the logs being generated behind the scene.

```
return chatClient.prompt() ChatClientRequestSpec
        .system( s: systemPrompt)
        .user( s: prompt)
        .advisors(
                ...advisors: new SimpleLoggerAdvisor()
        )
        .call() CallResponseSpec
        .content();
```

- Store the contents of a pdf and generate response to user's prompt from the pdf.
  - To read the pdf, spring-ai has a dependency

```
<dependency>
    <groupId>org.springframework.ai</groupId>
    <artifactId>spring-ai-pdf-document-reader</artifactId>
</dependency>
```

  - Everything under **src/main/resources/** directory are accessible via **ClassPathResource**

    my PDF path is **src/main/resources/spring_security.pdf**

```
@Value("classpath:static/spring_security.pdf")
Resource pdfFile;

public void storePdfData() {  1 usage
      ClassPathResource classPathResource = new ClassPathResource("static/spring_security.pdf");
      PagePdfDocumentReader pdfDocumentReader = new PagePdfDocumentReader(classPathResource);
    PagePdfDocumentReader pdfDocumentReader = new PagePdfDocumentReader( pdfResource: pdfFile);

    List<Document> pages = pdfDocumentReader.get();

    TokenTextSplitter tokenTextSplitter = TokenTextSplitter.builder()
            .withChunkSize( chunkSize: 200)
            .build();
    List<Document> chunks = tokenTextSplitter.apply( documents: pages);

    vectorStore.add(chunks);
}
```

    - **PagePdfDocumentReader** requires an object of type **Resource** inside its constructor.
    - Created a object of **ClassPathResource** (it implements **Resource**) and used it to create *pdfDocumentReader* object. Or you can get the object using **@Value** annotation.
    - **pdfDocumentReader.get()** method returns one chunk per page by default.
    - We need to create more chunks, so **TokenTextSplitter** can be used for this.
    - Then stored all those documents inside the database.
    - withCunkSize(200) doesn't mean it'll create 200 chunks, it means size of one chunk can be at max 200.

- To generate **chatClient** response, all the steps are same as before, I just changed the **similaritySearch** query a little bit

```
List<Document> documents = vectorStore.similaritySearch(
        searchRequest: SearchRequest.builder()
                .query( query: prompt)
                .topK( topK: 4)
                .similarityThreshold( threshold: 0.5)
                .filterExpression( textExpression: "file_name == 'spring_security.pdf'")
                .build()
);
```

-

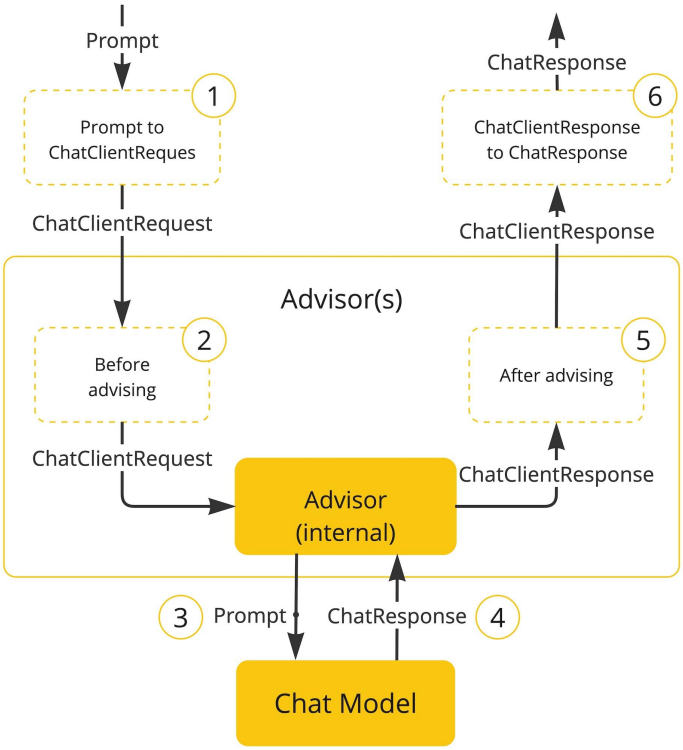
Classes & Objects used in the process of

- storing PDF data:
  - **PagePdfDocumentReader** to read the object.
  - **@Value("classpath:static/spring_security.pdf")** or **ClassPathResource** to get the resource.
  - **TokenTextSplitter** to create chunks with custom max chunk size.
  - **VectorStore** to store the data (embedding is done internally by itself).
- ask AI & generate response
  - **PromptTemplate** to create a template from a *String* with placeholders like **{context}** ---- required for system prompt.
  - **VectorStore** to perform the *similarity search* with **SearchRequest** object.
  - [creating system prompt from promptTemplate object by adding the *String* version of *documents* fetched from *vectorStore*]
  - **ChatClient** to get the response by passing *system prompt* and *user prompt*.
- F
-

- F
- F
- F
- F
-

Advisors

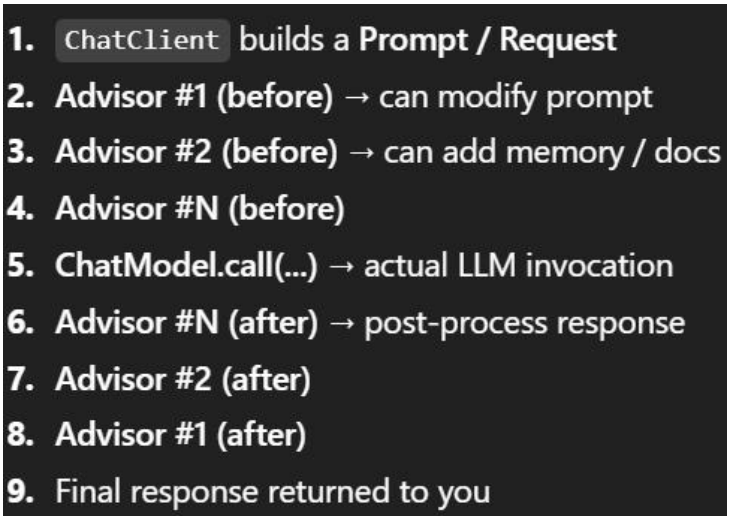
- Description
  - Advisors lies between **ChatClient** and **ChatModel**.



  -
    - prompt is sent, its sent to **ChatClient**.
    - Now it'll go through the ***advisors***, then it'll got o **ChatModel**.
    - After chat model gives response, it again comes via the ***advisors***.
  - **ChatClient** contains a method **prompt()** which returns the object of type **ChatClientRequestSpec**.
    - This **ChatClientRequestSpec** contains a method called **advisors**

```
ChatClientRequestSpec advisors(List<Advisor> var1);
```

  - **Advisors** are not AOP proxies, these are present in **ChatClient**'s execution pipeline.
    - These are executed each time the **call()** method (present inside ChatClientRequestSpec) is called.
    - Applied **before & after** the model is called.

- What actually happens:
  - 

  - The order of **advisors** in case of **ChatClient to ChatModel** are reversed in case of **ChatModel to ChatClient**.
- Some commonly used **Advisors**
  - **MessageChatMemoryAdvisor**
    - It makes the chat ***stateful*** by injecting the previous messages into the prompt.
    - Without this → every chat is ***stateless*** → AI will be DUMBBBBB.
    - It doesn't store the chats in vector store, it stores in a **ChatMemory** implementation.
  - **QuestionAnswerAdvisor** (RAG)
    - It fetches relevant information from the ***VectorStore*** and append those to the prompt.
    - Used in case of pdf, doc or something like that.
  - **VectorStoreChatMemoryAdvisor** (MOST COMMONLY USED)
    - Stores conversation history as **embeddings** inside ***VectorStore***, and retrieve only **relevant** past conversations.
    - ***MessageChatMemoryAdvisor*** remembers every conversation even those are not relevant, but ***VectorStoreChatMemoryAdvisor*** stores and get only relevant to current query.
  - **PromptChatMemoryAdvisor**
    - It injects predefined prompt memory like if you want to give any RULE to AI.
    - Example: system instructions, persona, rules
  - **SafeGuardAdvisor**

- It filters & blocks unwanted input/outputs.

- **Custom Logging Advisors**
  - Logs prompts, tokens, responses, timings.

- F
- **ChatMemory**
  - Most of the time you do not need to implement ChatMemory, everything is there already.
  - By default *InMemoryChatMemory* is provided by Spring AI.
  - It stores in JVM memory only.
  - If you want to implement Redis or something like that, then also implementations are there. You just need to **create bean of those**.
    - For *redis*: **RedisChatMemory**
  -

```
@Bean
ChatMemory chatMemory(RedisConnectionFactory factory) {
    return new RedisChatMemory(factory);
}
```

Using the Advisors

- You can set the default advisors globally while creating the bean of **ChatClient**.
  -

```
@Bean   Alok Ranjan Joshi *
public ChatClient chatClient( @NotNull ChatClient.Builder builder) {
    return builder
            .defaultAdvisors(
                    ...advisors: new SimpleLoggerAdvisor()
            )
    .build();
}
```

- Using **VectorStoreChatMemoryAdvisor**
  -

```
return chatClient.prompt() ChatClientRequestSpec
        .system( s: """
                You are an AI assistant called Cody.
                Greet users with your name (Cody) and the user name if you know their name.
                Answer in a friendly, conversational tone.
                """)
        .user( s: prompt)
        .advisors(
                ...advisors: VectorStoreChatMemoryAdvisor.builder( chatMemory: vectorStore)
                        .conversationId( conversationId: userId)
                        .defaultTopK( defaultTopK: 4)
                        .build()
        )
        .call() CallResponseSpec
        .content();
```

  - Here, **conversationId** is must, otherwise it'll not differentiate the different chats.
  - You can see the past conversations from long term memory.
    -

```
----------------------
LONG_TERM_MEMORY:
What is my name?
What is capital of India? and also, My name is Al
Hi Alok! I'm Cody. Your name is Alok. How can I a
Hi Alok! I'm Cody. The capital of India is New De
```

  - Also in database, you can see the entries (table: **vector_store**)
    -

-4fff-8f42-4892ffb6c88f	What is capital of India? and also, My name is Alok	{"messageType": "USE
433e-b2b8-9c6d1dbb6f31	Hi Alok! I'm Cody. The capital of India is New Delhi. If you have any	{"id": "chatcmpl-D2rjT
4d2a-9dae-e8e72a338cd1	What is my name?	{"messageType": "USE
-4ca4-9cdf-0116232e21d2	Hi Alok! I'm Cody. Your name is Alok. How can I assist you today?	{"id": "chatcmpl-D2rkF
449f-a44d-f91a72dcf617	What is my name? give me some good thoughts.	{"messageType": "USE
4eb6-8d12-838a64bdce20	Hi Alok! I'm Cody. It's great to chat with you! Here are some positiv	{"id": "chatcmpl-D2rl8

- In this case, irrespective of the time of the message, it'll fetch the most relevant message; but what if you want to get only last **N** messages.

- Using **MessageChatMemoryAdvisor**
  - It is a DUMB advisor, it keeps all the conversations and inject those all with the prompt.
  - But, how it gets all those messages?
    - It contains a **chat memory**, we can tell it to use any chat memory we want.
    - It executes **chatMemory.get(*conversationId*)** to get the messages and inject those.
    - What if we limit the number of messages in the **chat memory** itself, then we can achieve our goal to get only last **N** messages.
    - There is a class that implements **ChatMemory** and provides this feature: **MessageWindowChatMemory**
    - Whenever the advisor (MessageChatMemoryAdvisor) pushes new conversations, this chat memory (MessageWindowChatMemory) just stores that.
      - And when the advisor calls the **get()** method, it gives the latest **N** messages.
      - It doesn't remove old messages, it just gives the latest messages.
  - Creating bean of **MessageWindowChatMemory**

```
@Bean  new *
public ChatMemory chatMemory(JdbcChatMemoryRepository repository) {
    return MessageWindowChatMemory.builder()
            .chatMemoryRepository(repository)
            .maxMessages( maxMessages: 5)
            .build();
}
```

    - **JdbcChatMemoryRepository** is a chat memory repository.
    - **chat memory** is nothing but an abstraction that is used to store the data in **chat memory repository**.
    - **chat memory** has access to **chat memory repository** so that it can store and retrieve data from there.

```
.advisors(

        ...advisors: MessageChatMemoryAdvisor.builder(chatMemory)
                .conversationId( conversationId: userId)
                        .build(),

        VectorStoreChatMemoryAdvisor.builder( chatMemory: vectorStore)
                .conversationId( conversationId: userId)
                .defaultTopK( defaultTopK: 4)
                .build()
)
```

- chatMemory is nothing but the bean of *MessageWindowChatMemory*.
- This order is proper because, as the chat history is there in the prompt so VectorStoreChatMemoryAdvisor can search better.

- As we have used **JdbcChatMemoryRepository** here, so it'll store the messages in a table inside the database. If you want you can configure it to use *InMemoryChatMemory* which doesn't need any table in the database.

```
ai:
  ollama: {2 keys}
  openai: {3 keys}
  vectorstore: {1 key}
  chat:
    memory:
      repository:
        jdbc:
          initialize-schema: always
          table-name: chat_memory
```

- Order of advisors
  - The order of the advisors **matters**.
  - The updated prompt by the first advisor will become input context for 2nd advisor and so on.

- You can also implement **QuestionAnswerAdvisor** for RAG (any doc data)

➢ Creating custom Advisor

```java
public class TokenUsageAdvisor implements CallAdvisor {
    @Override  no usages
    public ChatClientResponse adviseCall(ChatClientRequest chatClientRequest, @NotNull CallAdvisorChain callAdvisorChain) {
        long startTime = System.currentTimeMillis();

        // Pass the request down the chain (to the LLM)
        ChatClientResponse advisedResponse = callAdvisorChain.nextCall(chatClientRequest);

        // Extract the actual LLM response
        ChatResponse chatResponse = advisedResponse.chatResponse();

        // inspect usage metadata
        if(chatResponse != null && chatResponse.getMetadata().getUsage() != null) {
            var usage = chatResponse.getMetadata().getUsage();
            long duration = System.currentTimeMillis() - startTime;

            log.info("Token usage: Inupt={} | Output={} | Total={} | Time={}ms",
                    usage.getPromptTokens(),
                    usage.getCompletionTokens(),
                    usage.getTotalTokens(),
                    duration);
        }

        return advisedResponse;
    }
```

➢

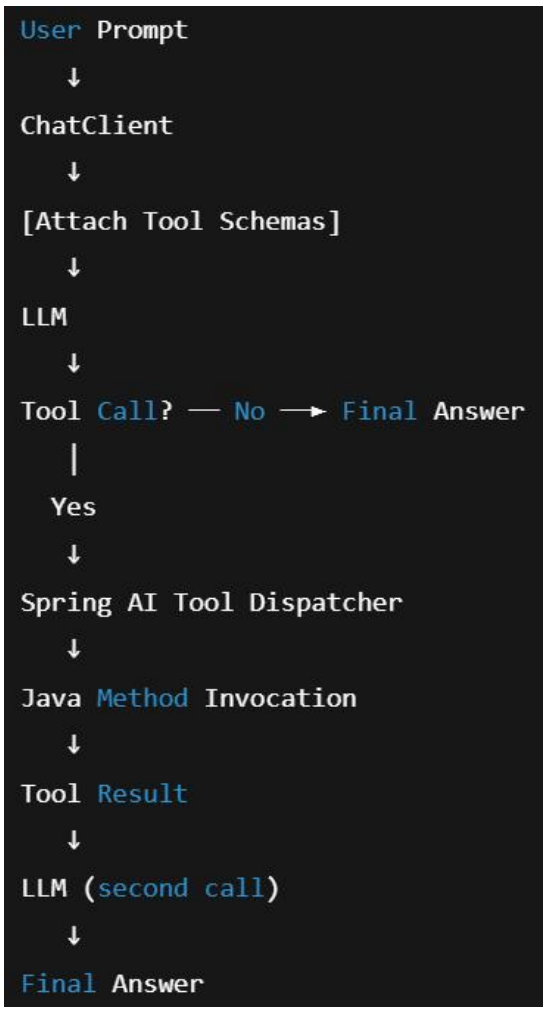
- Create a class (advisor) implementing the interface **CallAdvisor**
- It is a simple advisor just for logging purpose.
- **callAdvisorChain.nextCall()** will call take the control to the remaining advisors and get back here.
-

➢ F

➢ F

➢ F

Tool Calling



- Steps
  - You can create a tool with **@Tool** annotation, add this annotation to any method and it becomes a tool (provided the class that contains this method is an ***component***).

```
@Service
public class WeatherService {

    @Tool(description = "Fetch current temperature for a city")
    public Double getCurrentTemperature(String city) {
        // logic to call a weather API
        return weatherApi.lookup(city);
    }

}
```

  - You can give the **name**, **description** .

- After creating the tool, you need to attach that to any ***chatClient*** prompt()

```
String answer = chatClient.prompt("What's the weather in Paris?")
    .tools(weatherService)    // Making tools available to the model
    .call()
    .content();
```

  - Note, the bean is passed to ***tools()*** here.
  - You can give as many tools as you want here like
    - **tools(weatherService, locationService)**
- Now, if required, like if model doesn't find the response in its context, it'll make a tool call if a relevant tool is present.
  - It generates a structure like this

```
{
  "tool": "getCurrentTemperature",
  "arguments": {"city":"Paris"}
}
```

- Now, **Spring AI** *executes* your tool method with the provided arguments and returned value is **passed back to the *model***.
  - Now the Chat model will generate response according to that.

➢ Behind the Scenes

- 1. Spring search all methods annotated with **@Tools** and **reflect** those to get the data.

```
@Tool(description = "Get user by id")
public User getUser(Long id) { ... }
```

- 2. Each tool is converted into a ToolDefinition, which is a metadata structure.

```
ToolDefinition {
  name: "getUser"
  description: "Get user by id"
  inputSchema: {
    id: number
  }
  returnType: User
}
```

This is stored inside **ToolRegistry** inside Spring AI.

- When you call the ***chatClient*** call, Spring AI extracts the *tools methods* from the beans **which are attached to that chat client call** (only the attached), and create a structure and send that along with the prompt.

```
{
  "tools": [
    {
      "type": "function",
      "function": {
        "name": "getUser",
        "description": "Get user by id",
        "parameters": {
          "type": "object",
          "properties": {
            "id": { "type": "number" }
          },
          "required": ["id"]
        }
      }
    }
  ]
}
```

- Now, LLM sees 2 things: *user prompt, list of available tools + schemas*
  - It choose one of 2 paths: either a **simple text response** or **tool call**.
  - `"I cannot find the user"` **or**

```
{
  "tool_call": {
    "name": "getUser",
    "arguments": { "id": 10 }
  }
}
```

- If LLM makes tools call, Spring AI parses the model response and finds out that it is not a normal response, and call the tool with the given arguments by LLM.
  - It might be confusing that how Spring AI is so much intelligent to parse the LLM response that accurately?

    Because LLM is forced to send response in a fixed format if it wants tool call.

    Something like the below

- Now, after tool calling, Spring AI sends the response back to LLM again and LLM generate response.

- F
- F
- F
- F
- Ff
- F
- F
- F
- F
- F
-