- 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*

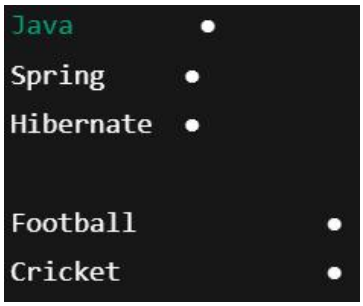
      without this, no chat with LLMs.
  - **spring-ai-model** (model contract layer)
    - Defines interfaces like **ChatModel**, **EmbeddingModel**
    - Makes providers interchangeable

      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

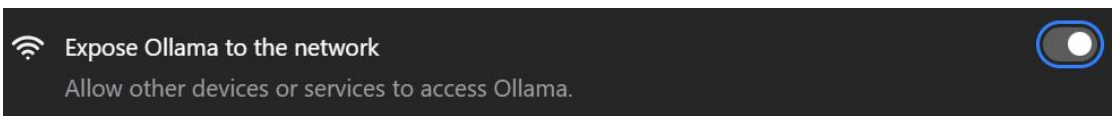
  ollama pull <AI model name>

  - ```
    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

-
- **F**
- **F**
- **F**
- **F**
- **F**
- **F**
- **F**

- Advisors
Spring AOP	Spring AI
Aspect	Advisor
Advice	Advisor logic
JoinPoint	Prompt / Response
Method call	LLM call
  - Spring AI advisors are NOT proxy-based
    They are explicit pipeline interceptors
Concept	Spring AI Advisor	Servlet Filter	Spring AOP Advice
Layer	LLM invocation pipeline	HTTP request pipeline	Method invocation pipeline
Entry point	`ChatClient`	`DispatcherServlet`	Proxy method call
Target	Prompt / Response	`HttpServletRequest/Response`	Method execution
Cross-cutting concern	Prompt enrichment, RAG, memory	Auth, logging, CORS	Tx, logging, security
Invocation style	**Explicit chain**	**Explicit chain**	**Proxy-based**
Ordering	Ordered list	Filter order	Aspect precedence
Can short-circuit?	✅ Yes	✅ Yes	✅ Yes
State aware?	✅ Yes	❌ Mostly stateless	❌ Mostly stateless
Provider aware?	❌ No	❌ No	❌ No
Uses proxies?	❌ No	❌ No	✅ Yes
- F
- F
- F
-