# How to Build a Conversational Assistant... Part 2: Supercharging Your Agent with an Embedding Store

Welcome back! In the first part of this series, you built a solid foundation for our "Super Insurance Company" chat agent. It had a personality and could hold a conversation, but let's be honest—it wasn't very smart about *our* business. It couldn't tell a customer a single thing about our actual insurance products.

So, how do we fix that? How do we give our agent a brain of its own, filled with our company's knowledge? That's exactly what Part 2 is all about.

This is where the magic happens. We're going to introduce a fundamental component of modern AI: the **Embedding Store**. Think of it as a specialized memory or knowledge base that our agent can search through in a very intelligent way. We will take our generic chatbot and make it a subject-matter expert on our fictional insurance products.

Here’s our game plan:

1. Briefly break down the concept of Embeddings and Embedding Stores.
2. Set up an in-memory store right within our Spring Boot application.
3. Load it with custom data about our insurance policies.
4. Upgrade our agent to use this new knowledge base to answer user questions accurately.

Ready to take your AI agent to the next level? Let's get started.

If you haven't read Part 1 of this series yet, you can find it here: <https://medium.com/@rodrigo.lopez.gatica/how-to-build-a-conversational-assistant-with-java-spring-boot-langchain4j-and-openai-0dab39f977dc>

## Step 0 – Our Starting Point: The Code from Part 1

First things first, let's refresh our memory on how our LangChainExampleRunner class looked at the end of Part 1. This is our starting point:

package com.superchat;

import dev.langchain4j.memory.chat.MessageWindowChatMemory;  
import dev.langchain4j.model.openai.OpenAiChatModel;  
import dev.langchain4j.service.AiServices;  
import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
import org.springframework.boot.CommandLineRunner;  
import org.springframework.context.annotation.Profile;  
import org.springframework.stereotype.Component;  
import java.time.Duration;  
import java.util.Scanner;  
@Component  
@Profile("langchain-example-1") // Run only if this profile is active  
public class LangChainExampleRunner implements CommandLineRunner {  
 private static final Logger logger = LoggerFactory.getLogger(LangChainExampleRunner.class);  
   
 @Override  
 public void run(String... args) throws Exception {  
 logger.info("LangChain4j Example Runner started...");  
 IChatAgentA chatAgentA = createAgenteChatRecomendador();  
  
 System.out.println("Type your question here:");  
 Scanner scanner = new Scanner(System.in);  
 while (true) {  
 System.out.print("You: ");  
 String line = scanner.nextLine();  
 if ("exit".equalsIgnoreCase(line) || "quit".equalsIgnoreCase(line)) {  
 break;  
 }  
 String respuesta = chatAgentA.chat(line);  
 System.out.printf("Agent response: %s%n", respuesta);  
 }  
 logger.info("LangChain4j Example Runner finished.");  
 }  
  
 private IChatAgentA createAgenteChatRecomendador() {  
 OpenAiChatModel chatModel = OpenAiChatModel.builder()  
 .apiKey("<your-api-key-here>")  
 .modelName("gpt-4o") // or "gpt-4-1106-preview", "gpt-3.5-turbo"  
 .temperature(0.2)  
 .timeout(Duration.ofSeconds(60))  
 .logRequests(false)  
 .logResponses(false)  
 .build();  
 IChatAgentA chatService = AiServices.builder(IChatAgentA.class)  
 .chatModel(chatModel)  
 .chatMemory(MessageWindowChatMemory.withMaxMessages(50))  
 .build();  
 return chatService;  
 }  
}

The core configuration for our simple assistant lives inside the createAgenteChatRecomendador() method. Here, we create an OpenAiChatModel using the "**gpt-4o**" model and then build an AiServices instance, linking it to our model and an OpenAI API Key. This service implements our Java interface, IChatAgentA.

Speaking of which, let's also remember what that IChatAgentA interface looked like:

package com.superchat;  
  
import dev.langchain4j.service.SystemMessage;  
import dev.langchain4j.service.UserMessage;  
  
public interface IChatAgentA {  
  
 @SystemMessage("""  
 You are an advisor for Super Insurance Company. Your role is to advise clients on their life and health insurance policies.  
 """)  
 String chat(@UserMessage String userMessage);  
}

Okay, with that fresh in our minds, let's dive into the changes that will allow us to enhance our Runner and connect our AiServices to an embedding store for our insurance products.

## Step 1 – Moving the API Key to an Environment Variable and Injecting It

It's a golden rule in software development: **never hardcode secrets like passwords or API keys directly in your source code**. In general, you should avoid including secrets in any file that gets committed to a source control repository like Git.

With that in mind, we'll start by modifying our project to:

1. Place the OpenAI API Key in a file named .env, which is commonly used to configure environment variables (and must not be uploaded to remote source control repositories).
2. Define a property named langchain4j.openai.chat-model.api-key in the application.properties file, located in the src/main/resources folder of our project. We will inject the value for this property from the .env file indicated in the previous step.
3. Inject the langchain4j.openai.chat-model.api-key property into a local variable in our LangChainExampleRunner.

Let's apply these changes.

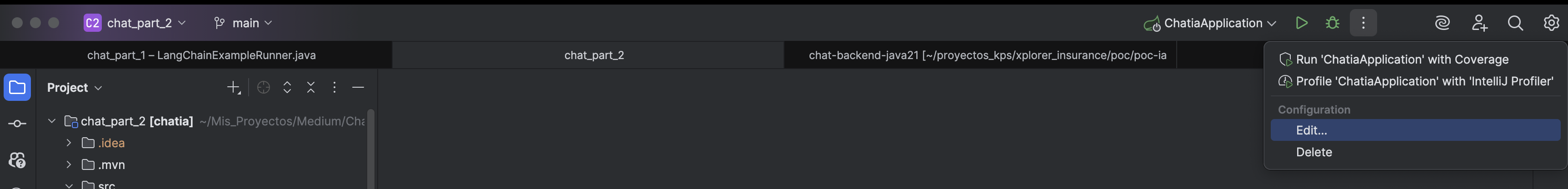
* 1. **Creating the .env file:**

In your IntelliJ IDE, navigate to the root directory of your Spring Boot project and create a new file named .env. Then, copy the following line into it:

OPENAI\_API\_KEY=<your-api-key-here>

Replace <your-api-key-here> with your actual OpenAI API Key.

To have your IDE process this file as a repository for environment variables, you'll need to click the “More actions” button and select the “Edit” option:



In the “**Run/Debug Configurations**” window, you need to locate the “**Environment variables**” text box. If it's not visible, select “**More options**”.



Then, in the pop-up menu, ensure that you have the “**Environment variables**” option selected.

A screenshot of a computer program

AI-generated content may be incorrect.

Now, in the “**Environment variables**” text box, you can either enter the full path to your .env file directly or click the “**Browse for .env files and scripts**” button to find the file.



Once the file path is entered, press the “**Apply**” button and then “**OK**”.

* 1. **Defining the langchain4j.openai.chat-model.api-key property in application.properties**

Open the application.properties file in your IntelliJ IDE and copy the following code:

spring.application.name=chatia

langchain4j.openai.chat-model.api-key=${OPENAI\_API\_KEY:}

The most important thing to note here is that on line 2, we've created a property called langchain4j.openai.chat-model.api-key whose value will be obtained from the OPENAI\_API\_KEY environment variable configured in the .env file.

It's crucial to point out that the expression ${OPENAI\_API\_KEY:} is what works this magic, and it **must** have the colon (:) after the environment variable name OPENAI\_API\_KEY.

* 1. **Injecting the langchain4j.openai.chat-model.api-key property into a local variable in our LangChainExampleRunner**

To finish handling the API Key, we will finally inject its value into a local variable within the Runner, which we'll call openAiApiKey. Here is the code block to perform the injection:

...

@Component

@Profile("langchain-example-1") // Run only if this profile is active

public class LangChainExampleRunner implements CommandLineRunner {

private static final Logger logger = LoggerFactory.getLogger(LangChainExampleRunner.class);

@Value("${langchain4j.openai.chat-model.api-key}") //Injects a variable with the value of the property "langchain4j.openai.chat-model.api-key" from application.properties

private String openAiApiKey;

...

The injection of the API Key's value into the openAiApiKey variable is achieved by annotating it with @Value("${langchain4j.openai.chat-model.api-key}").

## Step 2: Creating an Embedding Store for Our Company's Knowledge

So far, our conversational agent has a defined personality, but it lacks specific knowledge about "Super Insurance Company" products. To fix this, we need to give it access to a custom knowledge base. This is where one of the most powerful concepts in modern AI comes into play: the **Embedding Store**.

#### **What is an Embedding Store?**

Imagine an Embedding Store as a hyper-intelligent library. In a regular library, you search for books by title or author (keywords). In an Embedding Store, you search for information by **meaning or context**.

And here’s a key distinction: while the **heart** of this library is its **multidimensional numerical vectors**, the store actually holds several pieces of information to make everything work. For each piece of information we save, the Embedding Store holds three key components:

1. **The Embedding (The Multidimensional Numerical Vector):** This is the mathematical representation of the text's meaning. Think of it as an intelligent "index card" that allows the system to find contextually relevant information very quickly.
2. **The Content (The Original Text):** The original text segment that was used to generate the embedding. This is crucial because once the system finds the right "index card" (the embedding), it needs to be able to read the original text to use it as context.
3. **Metadata (Optional Additional Information): You can attach extra data to each segment, such as the source document's name, a date, a product category, etc. This can be useful for more advanced filtering (we'll cover how to use metadata for advanced filtering in a future installment)..**

#### **What Is It Used For?**

The main purpose of an Embedding Store in our project is to enable a technique called **Retrieval-Augmented Generation (RAG)** in a very simple way. When we connect our EmbeddingStore with AiServices, the flow is as follows:

1. The user asks a question (e.g., "Tell me about your life insurance for young people").
2. Langchain4j converts that question into an **embedding**.
3. **Automatically**, it searches the Embedding Store for the "closest" or most similar embeddings. This process of searching by meaning is **semantic search**. This isn't something we have to program manually in this step; AiServices handles it for us (**that said, in future installments of this series, we will explore how to perform these searches manually for more advanced control).**
4. Once the most relevant embeddings are found, it retrieves the **original text** associated with them.
5. Finally, it tells the LLM: "Using the following information: [the text retrieved from our documents], answer this question: [the user's original question]".

This way, the agent doesn't respond based on its generic knowledge but **bases its answer strictly on the information we provided**.

#### **Types of Embedding Stores in Langchain4j**

Langchain4j is very flexible and supports a wide variety of Embedding Stores, which can be grouped into two main categories:

1. **In-Memory:**
   * InMemoryEmbeddingStore: Stores everything (embeddings, text, and metadata) directly in the application's RAM. This is the one we will use in this article.
2. **Vector Databases (External and Persistent):**
   * These are robust solutions designed for production and large volumes of data. Langchain4j has integrations with the most popular ones, such as **Chroma**, **Pinecone**, **Weaviate**, **Milvus**, and extensions for traditional databases like **PGvector** for PostgreSQL and **Redis** (in a future installment, we'll integrate our solution with PostgreSQL 17 and its pgVector extension to manage a persistent embedding store).

#### **Our Choice:** InMemoryEmbeddingStore

For this article, we will use InMemoryEmbeddingStore. Why? Because it's the simplest and fastest way to get started.

* **What does it allow us to do?** It lets us create a functional knowledge base **without needing to install or configure any external database**. Everything lives inside our Spring Boot application.
* **Ideal for:** Prototypes, tests, demos, and applications with a small to medium amount of data.
* **Its main characteristic (and limitation):** It's **volatile**. This means that every time we restart the application, the Embedding Store is created empty, and we must load it with our documents again. For a production application, a persistent solution would be chosen, but for our tutorial, InMemoryEmbeddingStore is the perfect tool for learning the concepts.

#### **Creating and Populating Our Embedding Store**

Now that we understand the theory, it's time to put it into practice. We'll create a method in our LangChainExampleRunner class that will handle two key tasks:

1. Create an instance of our InMemoryEmbeddingStore.
2. Populate it with information about "Super Insurance Company's" products.

Here's the code that makes it possible:

public EmbeddingStore<TextSegment> createInMemoryEmbeddingStoreWithSampleData(EmbeddingModel embeddingModel) {

EmbeddingStore<TextSegment> embeddingStore = new InMemoryEmbeddingStore<>();

List<String> productList = new ArrayList<>();

productList.add(

"""

Individual Life Insurance: Insurance designed to provide financial protection to your loved ones in case of death.

- Coverages:

- Natural death: Provides a benefit for death due to natural causes.

- Accidental death: Covers death by accidents, offering an additional benefit.

""");

productList.add(

"""

Personal Accident Insurance: Insurance that offers protection in case of accidents resulting in injuries or death.

- Coverages:

- Accidental death: Provides a benefit for death due to accidents.

- Permanent disability: Covers permanent disability resulting from an accident, offering financial benefits.

""");

productList.add(

"""

Health Insurance: Insurance that covers medical expenses for illnesses or accidents.

- Coverages:

- Hospitalization: Covers costs of hospitalization due to illness or accident.

- Surgical procedures: Covers expenses for surgeries required due to health issues.

- Medical consultations: Provides coverage for medical consultations with specialists.

""");

for (String content : productList) {

Document doc = Document.from(content);

EmbeddingStoreIngestor.builder()

.embeddingModel(embeddingModel)

.embeddingStore(embeddingStore)

.build()

.ingest(doc);

}

logger.info("Product description loaded into embedding store.");

return embeddingStore;

}

#### **Let's Break Down the Code**

This method might look complex at first, but it's actually very logical. Let's break it down into pieces:

1. **Creating the 'Library':** The first line, new InMemoryEmbeddingStore<>(), creates an instance of our knowledge 'library'. At this point, it's completely empty.
2. **Preparing the Knowledge:** Next, we create a productList of type String. This is where we define the information about our fictional insurance products. Each productList.add() call adds a new 'document' that we want our agent to know about.
3. **The Ingestion Process (The Magic):** This is where the most interesting part happens. For each product description in our list, we do the following:
   * **We use the EmbeddingStoreIngestor:** Langchain4j provides this helper class, which acts like an intelligent 'librarian'. Its sole purpose is to take documents, process them, and file them away correctly in the Embedding Store.
   * **We give it instructions:** Using the .builder() pattern, we tell the 'ingestor' exactly what to do:
     + .embeddingModel(embeddingModel): "Use this AI model to read the document and understand its meaning (i.e., to create the multidimensional vector)."
     + .embeddingStore(embeddingStore): "And save the result in this 'library'."
   * **Ingest!**: The .ingest(doc) method executes the command. It takes our text, converts it into a Document, passes it to the EmbeddingModel to generate the vector, and finally stores both the original text and the resulting vector in our embeddingStore.

At the end of the process, the method returns the embeddingStore instance. It's no longer empty; instead, it's filled with all the specific knowledge about our products, ready to be queried.

## Step 3 – Upgrading Our Agent to Use the Embedding Store

We now have a method that creates a knowledge base—our EmbeddingStore—filled with our company's product information. The next logical step is to connect this knowledge base to our agent's "brain" so it can actually use it.

To do this, we need to update our createAgenteChatRecomendador() method. It will now be responsible for creating the embedding model, building the store, defining a mechanism to retrieve content, and finally, wiring everything together.

Here is the updated code:

private IChatAgentA createAgenteChatRecomendador() {

OpenAiChatModel chatModel = OpenAiChatModel.builder()

.apiKey(openAiApiKey)

.modelName("gpt-4o") // Or "gpt-4-1106-preview", "gpt-3.5-turbo"

.temperature(0.2)

.timeout(Duration.ofSeconds(60))

.logRequests(false)

.logResponses(false)

.build();

EmbeddingModel embeddingModel = OpenAiEmbeddingModel.builder()

.apiKey(openAiApiKey)

.modelName("text-embedding-3-small")

.dimensions(1536) // Important for text-embedding-3-small

.logRequests(false)

.logResponses(false)

.build();

EmbeddingStore<TextSegment> embeddingStore = createInMemoryEmbeddingStoreWithSampleData(embeddingModel);

EmbeddingStoreContentRetriever contentRetriever = EmbeddingStoreContentRetriever.builder()

.embeddingStore(embeddingStore)

.embeddingModel(embeddingModel)

.maxResults(7) // Number of fragments to retrieve.

.build();

IChatAgentA chatService = AiServices.builder(IChatAgentA.class)

.chatModel(chatModel)

.contentRetriever(contentRetriever)

.chatMemory(MessageWindowChatMemory.withMaxMessages(30))

.build();

return chatService;

}

#### **Let's Break It Down**

This updated method is the heart of our new, smarter agent. Let's walk through the new additions.

**1. Defining the EmbeddingModel**

First, we create an EmbeddingModel. It's crucial to understand that this is **different from our chatModel**.

* The chatModel (like gpt-4o) is a powerful "reasoning engine" designed to hold conversations.
* The embeddingModel (like text-embedding-3-small) is a specialized, highly efficient model whose only job is to convert text into numerical vectors (embeddings).
* We also specify the dimensions parameter, which is a requirement for certain models like text-embedding-3-small to ensure the vectors are created correctly.

**2. Creating and Populating the Store**

This line should look familiar! createInMemoryEmbeddingStoreWithSampleData(embeddingModel). We are now calling the method we created in Step 2 and passing the embeddingModel to it. This is because the "ingestor" inside that method needs the embedding model to convert our product descriptions into vectors before storing them.

**3. Introducing the EmbeddingStoreContentRetriever** 🔎

This is a new and vital component. If the EmbeddingStore is our library, the ContentRetriever is the **research assistant**. Its job is to:

1. Take the user's query.
2. Use the embeddingModel to understand the query's meaning.
3. Search the embeddingStore to find the most contextually relevant pieces of text.
4. Hand that relevant text over to the chat model.

We configure it with the embeddingStore (where to search) and the embeddingModel (how to search). We also set maxResults(7) to tell it to retrieve a maximum of the 7 most relevant text segments. This prevents flooding the chat model with too much information.

**4. Connecting Everything in AiServices (Enabling RAG)**

This is the final and most important piece of the puzzle. In our AiServices.builder(), we add a new line:

.contentRetriever(contentRetriever)

By adding this single line, we are officially enabling the **Retrieval-Augmented Generation (RAG)** pattern. We are telling AiServices to follow a new, smarter workflow: before you send anything to the chatModel, you **must first** use the contentRetriever to fetch relevant information from our knowledge base.

The agent's thought process is now: Query -> Retriever -> Relevant Context -> Chat Model -> Final Answer.

And that's it! Our createAgenteChatRecomendador method now builds a fully-fledged RAG agent that grounds its answers in the custom data we provided.

## Step 4 – Refining the System Prompt for Better Interaction

Our agent is now equipped with a knowledge base, but great conversational AI isn't just about having the right information—it's also about interacting with the user in a natural and effective way. The simplest and most powerful tool we have for controlling our agent's behavior is the @SystemMessage annotation.

Before we put all the code together, let's make a small but significant tweak to our IChatAgentA interface to guide the agent's initial interaction.

Here is the new version of the interface:

public interface IChatAgentA {

@SystemMessage(value =

"""

You are an advisor for the insurance company Super Insurance Company who must assist clients regarding their life and health insurance.

In your initial interaction with the user, ask for their full name.

""")

String chat(@UserMessage String userMessage);

}

#### **What Did We Change and Why?**

The change is subtle but impactful. We've added a single instruction to our system prompt:

"In your initial interaction with the user, ask for their full name."

By adding this line, we are giving our agent a new behavioral rule. We are not just telling it who to be (an advisor), but also what to do at the start of the conversation.

**Why is this important?**

1. **Personalization:** Asking for a name immediately makes the conversation feel more personal and less robotic, mimicking how a real human advisor would begin an interaction.
2. **Setting a Professional Tone:** This directive establishes a professional, client-focused workflow from the very first message.
3. **Foundation for Future Features:** While we aren't using the name for anything yet, this lays the groundwork for future enhancements. In a real-world application, you could use the name to retrieve a customer's profile from a database.

This small change demonstrates the power of prompt engineering. Without altering any other Java logic, we've refined our agent's behavior to be more effective and user-friendly.

With our knowledge base in place and our agent's behavior refined, all the pieces are now assembled. In the final step, we'll look at the complete code and recap everything we've built.

## Step 5 – Putting It All Together: The Final Code

We've walked through each new concept step-by-step: securing our API key, creating a knowledge base with an EmbeddingStore, and upgrading our agent with a ContentRetriever. Now, let's see how all these pieces fit together in our final LangChainExampleRunner.java class.

#### **The Complete Code**

Here is the complete, final version of our runner class for Part 2, which you just provided:

package com.superchat;

import dev.langchain4j.data.document.Document;

import dev.langchain4j.data.segment.TextSegment;

import dev.langchain4j.memory.chat.MessageWindowChatMemory;

import dev.langchain4j.model.embedding.EmbeddingModel;

import dev.langchain4j.model.openai.OpenAiChatModel;

import dev.langchain4j.model.openai.OpenAiEmbeddingModel;

import dev.langchain4j.rag.content.retriever.EmbeddingStoreContentRetriever;

import dev.langchain4j.service.AiServices;

import dev.langchain4j.store.embedding.EmbeddingStore;

import dev.langchain4j.store.embedding.EmbeddingStoreIngestor;

import dev.langchain4j.store.embedding.inmemory.InMemoryEmbeddingStore;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.beans.factory.annotation.Value;

import org.springframework.boot.CommandLineRunner;

import org.springframework.context.annotation.Profile;

import org.springframework.stereotype.Component;

import java.time.Duration;

import java.util.ArrayList;

import java.util.List;

import java.util.Scanner;

@Component

@Profile("langchain-example-2") // Note: I suggest changing the profile to example-2 for clarity

public class LangChainExampleRunner implements CommandLineRunner {

private static final Logger logger = LoggerFactory.getLogger(LangChainExampleRunner.class);

@Value("${langchain4j.openai.chat-model.api-key}")

private String openAiApiKey;

@Override

public void run(String... args) throws Exception {

logger.info("LangChain4j Example Runner started...");

IChatAgentA chatAgentA = createAgenteChatRecomendador();

System.out.println("Type your question here:");

Scanner scanner = new Scanner(System.in);

while (true) {

System.out.print("You: ");

String line = scanner.nextLine();

if ("exit".equalsIgnoreCase(line) || "quit".equalsIgnoreCase(line)) {

break;

}

String respuesta = chatAgentA.chat(line);

System.out.printf("Agent response: %s%n", respuesta);

}

logger.info("LangChain4j Example Runner finished.");

}

private IChatAgentA createAgenteChatRecomendador() {

// 1. Create the Chat Model (the "reasoning brain")

OpenAiChatModel chatModel = OpenAiChatModel.builder()

.apiKey(openAiApiKey)

.modelName("gpt-4o")

.temperature(0.2)

.timeout(Duration.ofSeconds(60))

.logRequests(false)

.logResponses(false)

.build();

// 2. Create the Embedding Model (the "meaning understanding" model)

EmbeddingModel embeddingModel = OpenAiEmbeddingModel.builder()

.apiKey(openAiApiKey)

.modelName("text-embedding-3-small")

.dimensions(1536)

.logRequests(false)

.logResponses(false)

.build();

// 3. Create and populate the Embedding Store (the "knowledge base")

EmbeddingStore<TextSegment> embeddingStore = createInMemoryEmbeddingStoreWithSampleData(embeddingModel);

// 4. Create the Content Retriever (the "research assistant")

EmbeddingStoreContentRetriever contentRetriever = EmbeddingStoreContentRetriever.builder()

.embeddingStore(embeddingStore)

.embeddingModel(embeddingModel)

.maxResults(7)

.build();

// 5. Build the AiServices (connecting all the pieces)

IChatAgentA chatService = AiServices.builder(IChatAgentA.class)

.chatModel(chatModel)

.contentRetriever(contentRetriever)

.chatMemory(MessageWindowChatMemory.withMaxMessages(30))

.build();

return chatService;

}

public EmbeddingStore<TextSegment> createInMemoryEmbeddingStoreWithSampleData(EmbeddingModel embeddingModel) {

EmbeddingStore<TextSegment> embeddingStore = new InMemoryEmbeddingStore<>();

List<String> productList = new ArrayList<>();

// ... (product list data is omitted for brevity but is included in the full code block)

productList.add(

"""

Individual Life Insurance: ...

""");

productList.add(

"""

Personal Accident Insurance: ...

""");

productList.add(

"""

Health Insurance: ...

""");

for (String content : productList) {

Document doc = Document.from(content);

EmbeddingStoreIngestor.builder()

.embeddingModel(embeddingModel)

.embeddingStore(embeddingStore)

.build()

.ingest(doc);

}

logger.info("Product description loaded into embedding store.");

return embeddingStore;

}

}

#### **What We've Accomplished**

Let's take a moment to recap everything we've achieved in this article. We have significantly evolved our simple chatbot by:

1. **Securing our Credentials:** We removed the hardcoded API key and now load it securely from an environment variable, a crucial best practice.
2. **Creating a Custom Knowledge Base:** We built an InMemoryEmbeddingStore and populated it with specific information about our company's products.
3. **Introducing a Specialized Embedding Model:** We added an OpenAiEmbeddingModel (text-embedding-3-small), a model designed specifically to understand the semantic meaning of our text.
4. **Building a Bridge to Knowledge:** We implemented an EmbeddingStoreContentRetriever, which acts as the intelligent assistant that searches our knowledge base for relevant information based on the user's query.
5. **Enabling RAG:** By connecting the ContentRetriever to our AiServices, we have officially transformed our agent. It no longer relies solely on its generic training; it now uses the **Retrieval-Augmented Generation (RAG)** pattern to provide context-aware, accurate answers.

Our agent is no longer just a friendly conversationalist—it's now a true, knowledgeable assistant for "Super Insurance Company."

## Step 6 – Putting Our New Agent to the Test

With all the improvements from this article now in place, let's run our Insurance Advisor Agent (Chatbot) project and see what happens.

Here is an example of a conversation I had with our newly upgraded agent:

**Type your question here:**

**You: Hi**

**Agent response: Hello! Thank you for reaching out to Super Insurance Company. Before we proceed, could you please provide me with your full name?**

**You: I'm Rodrigo López**

**Agent response: Thank you, Rodrigo López. How can I assist you today with your insurance needs? Are you interested in learning more about our health insurance, individual life insurance, or personal accident insurance options?**

**You: What life and health insurance does your company offer?**

**Agent response: At Super Insurance Company, we offer a range of life and health insurance options to meet your needs:**

**1. \*\*Individual Life Insurance\*\*: This insurance provides financial protection to your loved ones in the event of your death. It includes coverage for:**

**- \*\*Natural Death\*\*: Offers a benefit for death due to natural causes.**

**- \*\*Accidental Death\*\*: Provides an additional benefit for death resulting from accidents.**

**2. \*\*Personal Accident Insurance\*\*: This insurance offers protection in case of accidents that result in injuries or death. It includes coverage for:**

**- \*\*Accidental Death\*\*: Provides a benefit for death due to accidents.**

**- \*\*Permanent Disability\*\*: Offers financial benefits for permanent disability resulting from an accident.**

**3. \*\*Health Insurance\*\*: This insurance covers medical expenses for illnesses or accidents. It includes coverage for:**

**- \*\*Hospitalization\*\*: Covers the costs of hospitalization due to illness or accident.**

**- \*\*Surgical Procedures\*\*: Covers expenses for surgeries required due to health issues.**

**- \*\*Medical Consultations\*\*: Provides coverage for medical consultations with specialists.**

**If you have any specific questions or need further details about any of these options, feel free to ask!**

#### **Dissecting the Conversation: What Just Happened?**

This short conversation perfectly demonstrates the success of all the new features we added in this article. Let's break it down:

1. **The Proactive Greeting:** Notice the agent's very first response. It immediately asks for my full name. This isn't random; it's directly following the new instruction we gave it in the @SystemMessage in Step 4. Our prompt refinement worked perfectly!
2. **The RAG Payoff:** The most critical part of the conversation is when I ask, "What life and health insurance does your company offer?". A generic chatbot would have given a vague answer about what insurance usually is. Our agent, however, did something incredible:
   * It took my question and searched its **Embedding Store**.
   * It found the three product descriptions we fed it in Step 2.
   * It passed that specific information to the gpt-4o model as context.
3. **The Context-Aware Answer:** The agent's final response is a beautifully formatted summary of **only the products we defined**. It didn't invent anything or use outside knowledge. It used the data from our "knowledge base" to generate a precise, helpful, and accurate answer. This proves that our entire **Retrieval-Augmented Generation (RAG)** pipeline is fully functional.

Our chatbot has successfully evolved. It's now not only a polite conversationalist but a genuinely knowledgeable assistant for Super Insurance Company.