# 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>

## Paso 0 – Código fuente de la clase principal construida en la Parte 1 de esta entrega

Primero que nada, comencemos recordando como quedó la clase LangChainExampleRunner de nuestro Chatbot Asistente, que es donde comienza todo:

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;  
 }  
}

La configuración de nuestro simple asistente está en el método createAgenteChatRecomendador(), en donde creamos un OpenAiChatModel utilizando el modelo “gpt-4o” y luego creamos un AiServices al cual asociamos ese modelo y un API Key de OpenAI, y que devuelve una interfaz Java llamada IChatAgentA. Recordemos también como lucía esa interfaz IChatAgentA:

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);  
}

Con eso en mente, comencemos con las adaptaciones que nos permitirán mejorar nuestro Runner y vincular nuestro AiServices con un embedding store de productos de seguros.

## Paso 1 – Llevando el API Key a una variable de entorno que será inyectada en el archivo application.properties y a nuestro Runner.

Es una buena práctica evitar incluir en el código fuente los secretos tales como contraseñas y API Keys (y en general se debe evitar incluir secretos en cualquier archivo que luego deba ser subido a repositorios de control de código fuente, como Git).

Dado lo anterior, comenzaremos modificando nuestro proyecto para:

1. Dejar el API Key de OpenAI en un archivo llamado “.env” el cual suele ser usado para configurar **variables de entorno** (y que no debe subirse a los repositorios remotos de control de código fuente).
2. Definir una propiedad llamada langchain4j.openai.chat-model.api-key en el archivo application.properties, que se encuentra ubicado en la carpeta src/main/resources de nuestro proyecto. El valor de esa propiedad la inyectaremos desde el archivo .env que creamos en el paso anterior.
3. Inyectar la propiedad langchain4j.openai.chat-model.api-key en una variable local de nuestro LangChainExampleRunner.

Apliquemos esas modificaciones.

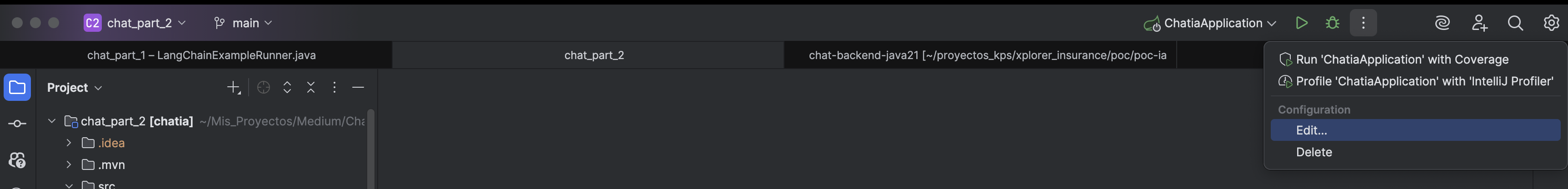
* 1. **Creación de archivo .env**:

En tu IDE IntelliJ posiciónate en el directorio raíz de tu proyecto Spring Boot y crea un archivo llamado .env. Luego copia la siguiente línea:

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

Reemplaza <yor-api-key-here> con tu API Key de OpenAI.

Para que este archivo sea procesado por tu IDE como repositorio de variables de entorno deberás clickear en el botón “More actions” y seleccionar la opción “Edit”:



En la ventana “Run/Debug Configurations” debes ubicar el cuadro de texto “Environment variables”. Si no aparece, entonces selecciona “More options”.



Luego, en el menú emergente, verifica que tengas seleccionada la opción “Environment variables”.

A screenshot of a computer program

AI-generated content may be incorrect.

Ahora en la caja de texto “Envionment variables” puedes ingresar directamente la ruta completa tu archivo “.env” o bien clickear el botón “Browse for .env files and scripts” y buscar el archivo.



Una vez ingresada la ruta del archivo, persiona el botón “Apply” y luego “OK”

* 1. **Definir la propiedad langchain4j.openai.chat-model.api-key en el application.properties**

Abre el archivo application.properties en tu IDE IntelliJ y copia el siguiente código:

spring.application.name=chatia

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

De acá lo más importante a señalar es que en la línea 2 hemos creado una propiedad llamada langchain4j.openai.chat-model.api-key cuyo valor será obtenido desde la variable de entorno OPENAI\_API\_KEY configurada en el archivo .env.

Es importante señalar que la expresión ${OPENAI\_API\_KEY:} es la que hace esta magia y que ésta debe tener el símbolo “:” luego del nombre de la variable de entorno OPENAI\_API\_KEY.

* 1. **Inyectar la propiedad langchain4j.openai.chat-model.api-key en una variable local de nuestro LangChainExampleRunner**

Para terminar con el manejo del AP Key, finalmente inyectaremos su valor en una variable local del Runner que llamaremos openAiApiKey. Acá va el bloque de código para hacer la inyección:

...

@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;

...

La inyección del valor del API Key en la variable openAiApiKey se logra anotándola con @Value("${langchain4j.openai.chat-model.api-key}").