# Introducción

Han pasado varios meses desde que hice la última actualización de la serie “**How to Build a Conversational Assistant with Java Spring Boot, Langchain4j and OpenAI**”. Y es que he estado inmerso en un proyecto de desarrollo e implementación de una nueva plataforma digital para Bancaseguros para una importante compañía de seguros de Panamá; un gran cliente nuestro con gente muy buena.

Sin embargo, ahora que ya pusimos en producción la nueva plataforma, he vuelto para retomar esta serie.

A modo de un breve repaso, en las entregas anteriores de esta serie logramos construir un **agente recomendador de seguros**, bastante simple, conectado a una **base de datos vectorial (embeddings) implementada en PostgreSQL**.

El *espíritu* de este **agente conversacional recomendador de productos de seguros es** **ofrecer los productos más apropiados al perfil del cliente**, solicitando para esto información de su **edad**, su **estado civil**, **hijos**, etc., todo a través de una conversación lo más natural posible (y sin utilizar esos menús de opciones “digite 1 para esto, 2 para esto otro…”, como es común ver en chats de “la vieja escuela”).

Ahora, tal como lo prometí hace meses atrás, seguiremos profundizando y afinando nuestro **Agente Recomendador** agregándole **nuevas capacidades como búsqueda semántica**, **filtros de búsqueda basados en metadata** e **invocación de herramientas para cotizar seguros** (cálculo de prima).

Esta nueva entrega quizá tenga un **nivel de complejidad técnica mayor a las entregas anteriores**. Sin embargo, iremos paso a paso y les mostraré cada fragmento de código fuente para que puedan replicar este agente recomendador en sus entornos locales.

Describiremos esta nueva versión del “Agente Recomendador Potenciado” a través de la siguiente metodología:

1. Primero mostraré el código fuente (algoritmo) del **Runner** principal que **ejecuta el loop de interacción con el usuario/cliente**, el cual lo he dividido en **7 pasos conceptuales**: Step 1 – Setup; Step 2 - Manage Products; Step 3 - Start interaction loop; Step 4 - Get client profile; Step 5 - Find candidate products; Step 6 - Build context for main agent; Step 7 - Response to customer.
2. Por cada paso conceptual, explicaré qué hace el algoritmo y profundizaré en las clases utilizados en dichos pasos (servicios, repositorios, utilitarios, etc.).
3. Al final mostraré una nueva versión del archivo pom.xml, en donde podrán ver la configuración de librerías de Langchain4j, pgVector, etc., actualizadas a las versiones más recientes a la fecha (fines de Octubre de 2025).

¿Están listos? ¡Comencemos!

# 1. Runner Principal (algoritmo de interacción con el cliente)

|  |
| --- |
| package com.superchat;  import com.superchat.services.ProductIngestionService; import com.superchat.repositories.ProductRepository; import com.superchat.interfaces.IChatAgentA; import com.superchat.interfaces.IProfileExtractionAgent; import com.superchat.model.ClientProfile; import com.superchat.model.Product; import com.superchat.model.ProductRecommendationResult; import com.superchat.services.AudienceSearcherService; import com.superchat.services.IABuilderService; import com.superchat.utils.AgentContextBuilder; import dev.langchain4j.data.segment.TextSegment; import dev.langchain4j.model.embedding.EmbeddingModel; import dev.langchain4j.store.embedding.\*; import lombok.extern.slf4j.Slf4j; import org.springframework.boot.CommandLineRunner; import org.springframework.context.annotation.Profile; import org.springframework.stereotype.Component; import java.util.\*;  @Component @Profile("langchain-example-1") // Run only if this profile is active @Slf4j public class LangChainExampleRunner implements CommandLineRunner {  private final Boolean firstRun = false; // Flag to control data ingestion  private final IABuilderService iaBuilderService;  private final AudienceSearcherService audienceSearcher;  private final ProductIngestionService productIngestionService;   // Step 0 - Constructor-based dependency injection  public LangChainExampleRunner(IABuilderService iaBuilderService,  AudienceSearcherService audienceSearcher,  ProductIngestionService productIngestionService) {  this.iaBuilderService = iaBuilderService;  this.audienceSearcher = audienceSearcher;  this.productIngestionService = productIngestionService;  }   @Override  public void run(String... args) throws Exception {  *log*.info("LangChain4j Example Runner started...");  ClientProfile clientProfile = new ClientProfile();  String jsonDataForProfileExtractionAgent = null;  List<String> candidateIds = null;  String contextForAgent = "";   // Step 1 - Setup: Build the models and the store:  final ProductRecommendationResult productRecommendationResult = iaBuilderService.createAgenteChatRecomendador();   IChatAgentA chatAgentA = productRecommendationResult.getChatAgentA();  EmbeddingModel embeddingModel = productRecommendationResult.getEmbeddingModel();  EmbeddingStore<TextSegment> embeddingStore = productRecommendationResult.getEmbeddingStore();  IProfileExtractionAgent profileExtractionAgent = productRecommendationResult.getProfileExtractionAgent();   // Step 2 - Manage Products  // 2.1. Get the insurance products:  List<Product> products = ProductRepository.*findAllProducts*();   // 2.2. If first run, ingest products into the embedding store:  if (firstRun)  // Ingest only on first run  productIngestionService.ingestAll(products, embeddingStore, embeddingModel);   // Step 3 - Start interaction loop:  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) || "bye".equalsIgnoreCase(line)) {  *log*.info("LangChain4j Example Runner finished.");  break;  }   if (line == null || line.trim().isEmpty()) {  continue; // Skip empty lines  }   // Step 4 - Get client profile. When a message arrives from the user, try yo extract profile info using  // the profile extraction agent:  jsonDataForProfileExtractionAgent = profileExtractionAgent.extractData(line);  clientProfile.applyJson(jsonDataForProfileExtractionAgent);   *log*.info("Extracted client profile: {}", clientProfile.toString());  *log*.info("Friendly description: {}", clientProfile.friendlyProfileDescription());   // Step 5 - Find candidate products. Use AudienceSearcher to find candidate products based on the extracted  // profile:  candidateIds = audienceSearcher.findCandidateProductIds(  clientProfile,  embeddingModel,  embeddingStore,  7, // maxResults  0.75 // minScore  );   // Step 6 - Build context for main agent.  // The context will include only the candidate products... Not ALL products!:  // This improves performance and reduces costs.  contextForAgent = AgentContextBuilder.*buildContextForAgent*(products, candidateIds);   // Step 7 - Response to customer. Call main chat agent with context and user message, to get and appropriate  // response/offer for the customer:  String respuesta = chatAgentA.chat(line, contextForAgent);  System.*out*.printf("Agent response: %s%n", respuesta);  }  } } |

## Step 0

El runner comienza definiendo un constructor el cual es utilizado por Spring Boot para inyectar los servicios (clases anotadas con @Services, las cuales serán administradas por Spring Boot).

## Step 1

Acá comienza realmente lo interesante.

El método createAgenteChatRecomendador() del servicio IABuilderService se encarga de inicializar los objetos de **Langchain4j** que utilizaremos en el resto del algoritmo, que son:

* **EmbeddingModel**: es el **componente que convierte texto en vectores numéricos** (embeddings). Es decir: toma un String (o un TextSegment) y devuelve un **vector de dimensión fija** que captura el “significado” semántico del texto. Esos vectores luego se guardan en el **EmbeddingStore** (PostgreSQL + su extensión pgVector) y se comparan entre sí con una métrica de similitud/distancia para hacer **búsqueda semántica**. ¿**Para qué sirve?**
  + Antes de ingestar productos de seguros en la BD vectorial, el EmbeddingModel se utiliza para transforma descripciones de productos y sus audiencias en vectores.
  + Antes de realizar ejecutar la consulta semántica, el EmbeddingModel se utiliza paratransforma la “query” (ya sea el texto ingresado por el cliente o el texto que describe el perfil del cliente) en un vector que luego se envía al método search() del EmbeddingStore. Cabe señalar que utilizaremos el modelo “text-embedding-3-small”, el cual tiene una **dimensión/tamaño de 1536**. Es importante notar que este tamaño **debe estar alineado con la dimensión vectorial utilizada en la tabla de embeddings de PostgreSQL** (para más detalle respecto a la definición de la tabla de embeddings ver Parte 3 de esta serie).
* **EmbeddingStore<TextSegment>**: es la abstracción (interfaz) que guarda y consulta embeddings junto con su payload asociado. Piensa en él como un “índice vectorial” donde:
  + **Clave**: un vector (embedding) numérico (p. ej., 1536 dimensiones).
  + **Valor/payload**: el TextSegment que generó ese vector (tu texto + Metadata).
  + **¿Qué guarda exactamente el EmbeddingStore?**: El embedding (vector) del texto; el contenido (texto) que fue embebido; Metadata: pares clave/valor (como productId, segmentType=audience|details, ageMin/ageMax, category, etc.). En pgvector típicamente la tabla subyascente incluye las siguientes columnas: id, vector, texto, metadatos (JSON) y timestamps (la implementación de langchain4j se encarga del esquema de la tabla).
* **IProfileExtractionAgent**: técnicamente es una interfaz anotada con @SystemMessage (del framework Langchain4j) que fija el comportamiento del agente (formato, idioma, restricciones) encargado de construir el perfil del cliente. Este agente se encarga de obtener **un texto estructurado (JSON) con el perfil de cliente** a partir del texto libre que escribe el usuario. Ese JSON posteriormente es utilizado para cargar el objeto ClientProfile (ver más adelante para más detalles). **Este agente es un componente clave para el propósito de nuestro chat recomendador inteligente**.
* **IChatAgentA**: similar al el caso anterior, técnicamente es una interfaz anotada con @SystemMessage que fija el comportamiento del agente encargado de mantener una conversación con el cliente y recomendarle productos.

A continuación explicaré en detalle cada paso llevado a cabo por la clase “Runner”:

# Step 1 – Setup

// Step 1 - Setup: Build the models and the store:  
final ProductRecommendationResult productRecommendationResult = iaBuilderService.createAgenteChatRecomendador();  
  
IProfileExtractionAgent profileExtractionAgent = productRecommendationResult.getProfileExtractionAgent();  
IChatAgentA chatAgentA = productRecommendationResult.getChatAgentA();  
EmbeddingModel embeddingModel = productRecommendationResult.getEmbeddingModel();  
EmbeddingStore<TextSegment> embeddingStore = productRecommendationResult.getEmbeddingStore();

La aplicación comienza llamando al método createAgenteChatRecomendador() del service ProductRecommendationResult para obtener los objetos específicos de Langchain4j (incluyendo los agentes/servicios de IA) que se utilizarán en el resto del algoritmo de recomendación de productos.

## Step 2 - Manage Products

// Step 2 - Manage Products  
// 2.1. Get the insurance products:  
List<Product> products = ProductRepository.*findAllProducts*();  
  
// 2.2. If first run, ingest products into the embedding store:  
if (firstRun)  
 // Ingest only on first run  
 productIngestionService.ingestAll(products, embeddingStore, embeddingModel);

Acá simplemente invocamos al método findAllProducts() para obtener la lista completa de Productos y luego, si el flag firstRun es true, invocamos al servicio que se encarga de ingestarlos en la BD vectorial (productIngestionService.ingestAll()).

Por simplicidad, lo que yo hago es establecer firstRun a true sólo la primera vez que ejecuto la aplicación para permitir la ingesta de productos. Después de eso la establezco en false y la dejo así para siempre ☺.

Revisemos con más detalle qué hace el método ingestAll:

@Service  
@Slf4j  
public final class ProductIngestionService {  
  
 private ProductIngestionService(){}  
  
 public void ingestAll(List<Product> products,  
 EmbeddingStore<TextSegment> store,  
 EmbeddingModel embeddingModel) {  
 for (Product p : products) {  
 *log*.info("Ingesting product: {} - {}", p.id(), p.name());  
 ingestProduct(p, store, embeddingModel);  
 }  
 *log*.info("Ingestion completed for {} products.", products.size());  
 }  
  
 public void ingestProduct(Product p,  
 EmbeddingStore<TextSegment> store,  
 EmbeddingModel embeddingModel) {  
  
 // ===== AUDIENCE Segment =====  
 Metadata audMd = new Metadata();  
 audMd.put(*META\_PRODUCT\_ID*, p.id());  
 audMd.put(*META\_NAME*, p.name());  
 audMd.put(*META\_AGE\_MIN*, p.ageMin());  
 audMd.put(*META\_AGE\_MAX*, p.ageMax());  
 audMd.put(*META\_CATEGORY*, p.category());  
 audMd.put(*META\_SEGMENT\_TYPE*, *SEG\_AUDIENCE*);  
  
 String audienceText = "Target Audience: " + p.audienceText().trim();  
 TextSegment audienceSeg = TextSegment.*from*(audienceText, audMd);  
  
 Embedding audienceEmb = embeddingModel.embed(audienceSeg).content();  
 store.add(audienceEmb, audienceSeg);  
  
 // ===== DETAILS segment =====  
 Metadata detMd = new Metadata();  
 detMd.put(*META\_PRODUCT\_ID*, p.id());  
 detMd.put(*META\_NAME*, p.name());  
 detMd.put(*META\_AGE\_MIN*, p.ageMin());  
 detMd.put(*META\_AGE\_MAX*, p.ageMax());  
 detMd.put(*META\_CATEGORY*, p.category());  
 detMd.put(*META\_SEGMENT\_TYPE*, *SEG\_DETAILS*);  
  
 String detailsText = """  
 Product ID: %s  
 Product Name: %s  
 Product Description: %s  
 Coverages:  
 %s  
 Target Audience: %s  
 """.formatted(  
 p.id(), p.name(), p.description().trim(), p.coveragesText().trim(), p.audienceText().trim());  
  
 TextSegment detailsSeg = TextSegment.*from*(detailsText, detMd);  
  
 Embedding detailsEmb = embeddingModel.embed(detailsSeg).content();  
 store.add(detailsEmb, detailsSeg);  
 }  
}

Como se puede observar, el método se divide en dos grandes partes:

1. Ingesta del segmento de texto “Target Audience”, el cual contiene una descripción del “cliente objetivo” de un producto específico.

# Actualización de librerías

Como pasó bastante tiempo desde la publicación de la parte 3 de esta serie, comenzaremos presentando una nueva versión del archivo pom.xml, con una actualización de las versiones de las librerías:

## Archivo pom.xml

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?> <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">  <modelVersion>4.0.0</modelVersion>  <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>3.5.6</version>  <relativePath/> <!-- lookup parent from repository -->  </parent>  <groupId>com.superchat</groupId>  <artifactId>chatia</artifactId>  <version>0.0.1-SNAPSHOT</version>  <name>chatia</name>  <description>Mi primer Chat IA con LangChain4j y OpenIA</description>   <properties>  <java.version>24</java.version>  <langchain4j.version>1.7.1</langchain4j.version>  <langchain4j-pgvector.version>1.3.0-beta9</langchain4j-pgvector.version>  <langchain4j-spring-boot-starter.version>1.7.1-beta14</langchain4j-spring-boot-starter.version>  </properties>   <dependencies>   <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-data-jpa</artifactId>  </dependency>   <dependency>  <groupId>dev.langchain4j</groupId>  <artifactId>langchain4j-spring-boot-starter</artifactId>  <version>${langchain4j-spring-boot-starter.version}</version>  </dependency>   <dependency>  <groupId>dev.langchain4j</groupId>  <artifactId>langchain4j</artifactId>  <version>${langchain4j.version}</version>  </dependency>   <dependency>  <groupId>dev.langchain4j</groupId>  <artifactId>langchain4j-open-ai</artifactId>  <version>${langchain4j.version}</version>  </dependency>   <dependency>  <groupId>dev.langchain4j</groupId>  <artifactId>langchain4j-pgvector</artifactId>  <version>${langchain4j-pgvector.version}</version>  </dependency>   <dependency>  <groupId>org.postgresql</groupId>  <artifactId>postgresql</artifactId>  <scope>runtime</scope>  </dependency>   <dependency>  <groupId>org.projectlombok</groupId>  <artifactId>lombok</artifactId>  <optional>true</optional>  </dependency>   <!-- ADDED: Dependency for Testing -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-test</artifactId>  <scope>test</scope>  </dependency>  </dependencies>   <build>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin>  </plugins>  </build>  </project> |

## Repositorio de productos

En aras de la simplicidad simularemos un repositorio de productos, el cual cargaremos en memoria en un List (en una versión más evolucionada, ésta clase podría extender por ejemplo de un JpaRepository de Spring Boot y la clase Product podría ser un Entity JPA. Pero por ahora, en aras de la simplicidad, lo dejaremos como una lista simple de productos en memoria):

|  |
| --- |
| package com.superchat.repositories;  import com.superchat.model.Product;  import java.util.ArrayList; import java.util.List;  public final class ProductRepository {   private ProductRepository(){}   public static List<Product> findAllProducts() {  List<Product> items = new ArrayList<>();   items.add(new Product(  "PROD\_01",  "Individual Life Insurance",  "Insurance designed to provide financial protection to your loved ones in case of death.",  """  - Natural death: Provides a benefit for death due to natural causes.  - Accidental death: Covers death by accidents, offering an additional benefit.  """,  """  Adults aged 25-65, of any gender, who are primary income earners or have financial dependents  (such as spouses, children, or elderly parents), seeking to ensure the financial security and  well-being of their families in the event of unforeseen circumstances.  """,  25, 65, "life"  ));   items.add(new Product(  "PROD\_02",  "Personal Accident Insurance",  "Insurance that offers protection in case of accidents resulting in injuries or death.",  """  - Accidental death: Provides a benefit for death due to accidents.  - Permanent disability: Covers permanent disability resulting from an accident, offering financial  benefits.  """,  """  Adults aged 25-65, of any gender, who are exposed to risks of accidents in their daily activities,  such as workers, students, athletes, or people who frequently travel, and who wish to protect themselves  and their families from the financial consequences of accidental injuries or death.  """,  25, 65, "accident"  ));   items.add(new Product(  "PROD\_03",  "Health Insurance",  "Insurance that covers medical expenses for illnesses or accidents.",  """  - 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.  """,  """  Individuals and families of all ages (18-120) who are concerned about potential medical expenses due to  illness or accidents, including those with pre-existing health conditions, self-employed professionals,  parents seeking coverage for their children, elderly individuals, and anyone who wants to ensure access  to quality healthcare and financial protection against unexpected medical costs.  """,  18, 120, "health"  ));   items.add(new Product(  "PROD\_04",  "Young Adult Travel Insurance",  """  A comprehensive travel insurance plan designed for young adults who seek adventure, exploration, and  peace of mind while traveling. It offers essential protection against unexpected events that may  occur during domestic or international trips, allowing you to focus on enjoying your journey without  worries.  """,  """  - Medical emergencies abroad: Covers medical expenses resulting from illness or accidents during your trip.  - Trip cancellation or interruption: Provides reimbursement for non-refundable expenses if your trip is  canceled or cut short due to covered reasons.  - Lost or delayed baggage: Compensates for lost, stolen, or significantly delayed luggage.  - Travel assistance services: Offers 24/7 support for emergencies, including medical evacuation, legal  assistance, and travel advice.  """,  """  Young single adults aged 18–35, of any gender, who travel for leisure, study, or work and seek reliable  protection against travel-related risks. Ideal for frequent travelers, digital nomads, students studying  abroad, or professionals on business trips who value safety, flexibility, and peace of mind while  exploring the world..  """,  18, 35, "travel"  ));   items.add(new Product(  "PROD\_05",  "Pets Insurance",  "Insurance that covers medical expenses for illnesses or accidents of your loved pet.",  """  - 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.  """,  """  Oriented to people of all ages (18-120), owners of pets such as dogs and cats, who want to provide them  with protection against diseases.  """,  18, 120, "pet"  ));   items.add(new Product(  "PROD\_06",  "Home Insurance",  "Insurance that protects your home’s structure and contents against covered events and includes personal liability coverage.",  """  - Fire and smoke: Covers damage to the dwelling and contents caused by fire or smoke.  - Theft and vandalism: Covers stolen belongings and damage from forced entry or malicious acts.  - Water damage (sudden/accidental): Covers damage from burst pipes or appliance leaks (non-gradual).  - Natural events: Windstorm and hail; earthquake/flood available via optional endorsements.  - Glass breakage and fixtures: Covers windows, sanitary ware, and fixed installations.  - Temporary accommodation (loss of use): Pays for lodging if the home becomes uninhabitable due to a covered loss.  - Personal liability: Covers injuries to third parties or damage to their property caused by the insured household.  """,  """  Adults who own or rent a house or apartment and want financial protection for their dwelling, belongings, and liability.  Ideal for first-time homeowners, families, and landlords seeking comprehensive home coverage.  """,  21, 75, "home"  ));   items.add(new Product(  "PROD\_07",  "Car Insurance",  "Insurance that protects your car and your liability arising from its use, covering damage, theft, third-party claims, and roadside emergencies.",  """  - Third-party liability (bodily injury/property damage): Covers injuries to others and damage to their property caused by your car.  - Collision: Pays for repairs to your car after a crash, regardless of fault (subject to deductible).  - Comprehensive: Covers non-collision losses (theft, fire, vandalism, falling objects, weather events).  - Medical payments / personal injury protection: Covers medical expenses for you and your passengers after an accident.  - Uninsured/underinsured motorist: Protects you if the at-fault driver has insufficient or no insurance.  - Roadside assistance & towing: Help for breakdowns, flat tires, dead batteries, and emergency towing.  - Glass coverage: Repairs or replaces damaged windshields and windows.  - Rental car / mobility allowance: Provides a temporary vehicle while yours is being repaired after a covered loss.  - Optional accessories & custom parts: Extends coverage to added equipment (sound systems, racks, custom wheels).  """,  """  Licensed drivers who own or lease a car and want financial protection for their vehicle and liability.  Ideal for commuters, families, and everyday drivers; optional endorsements available for ride-share or delivery use.  """,  18, 75, "auto"  ));   return items;  } } |

Son **7 productos** de distinta categoría. De manera similar a las entregas anteriores, cabe recalcar que cada producto tiene un **ID**, un **nombre**, una **descripción** general del producto, una **descripción de las coberturas de riesgo** y una propiedad que indica la **audiencia objetivo**. Para nuestro propósito, **ésta última es una de las propiedades más importantes del producto ya que nuestro Agente Recomendador intentará recomendar productos principalmente en base al perfil del cliente comparándola semánticamente con las audiencias objetivo de los productos**. Lo explicaremos con mayor detalle más adelante.

Además, y a diferencia de lo desarrollado en las partes anteriores de esta serie, he incorporado tres propiedades adicionales que nos permitirán realizar filtros específicos: **edad mínima**, **edad máxima** y **categoría**.

## Registro Product

La lista anterior utiliza una *registro* (similar a un POJO) llamado **Product**:

|  |
| --- |
| package com.superchat.model;  public record Product(  String id,  String name,  String description,  String coveragesText,  String audienceText,  int ageMin,  int ageMax,  String category ) {} |

# Ingesta de productos en la BD vectorial

La ingesta de productos en la BD PostgreSQL (con su extensión pgVector) queda a cargo del siguiente Service (acá, a través de la incorporación de un @Service, comenzamos a utilizar capacidades específicas del framework Spring Boot):

|  |
| --- |
| package com.superchat.services;  import com.superchat.model.Product; import dev.langchain4j.data.document.Metadata; import dev.langchain4j.data.segment.TextSegment; import dev.langchain4j.data.embedding.Embedding; import dev.langchain4j.model.embedding.EmbeddingModel; import dev.langchain4j.store.embedding.EmbeddingStore; import lombok.extern.slf4j.Slf4j; import org.springframework.stereotype.Service;  import java.util.List;  import static com.superchat.services.ProductFieldsUtil.\*;  @Service @Slf4j public final class ProductIngestionService {   private ProductIngestionService(){}   public void ingestAll(List<Product> products,  EmbeddingStore<TextSegment> store,  EmbeddingModel embeddingModel) {  for (Product p : products) {  *log*.info("Ingesting product: {} - {}", p.id(), p.name());  ingestProduct(p, store, embeddingModel);  }  *log*.info("Ingestion completed for {} products.", products.size());  }   public void ingestProduct(Product p,  EmbeddingStore<TextSegment> store,  EmbeddingModel embeddingModel) {   // ===== AUDIENCE Segment =====  Metadata audMd = new Metadata();  audMd.put(*META\_PRODUCT\_ID*, p.id());  audMd.put(*META\_NAME*, p.name());  audMd.put(*META\_AGE\_MIN*, p.ageMin());  audMd.put(*META\_AGE\_MAX*, p.ageMax());  audMd.put(*META\_CATEGORY*, p.category());  audMd.put(*META\_SEGMENT\_TYPE*, *SEG\_AUDIENCE*);   String audienceText = "Target Audience: " + p.audienceText().trim();  TextSegment audienceSeg = TextSegment.*from*(audienceText, audMd);   Embedding audienceEmb = embeddingModel.embed(audienceSeg).content();  store.add(audienceEmb, audienceSeg);   // ===== DETAILS segment =====  Metadata detMd = new Metadata();  detMd.put(*META\_PRODUCT\_ID*, p.id());  detMd.put(*META\_NAME*, p.name());  detMd.put(*META\_AGE\_MIN*, p.ageMin());  detMd.put(*META\_AGE\_MAX*, p.ageMax());  detMd.put(*META\_CATEGORY*, p.category());  detMd.put(*META\_SEGMENT\_TYPE*, *SEG\_DETAILS*);   String detailsText = """  Product ID: %s  Product Name: %s  Product Description: %s  Coverages:  %s  Target Audience: %s  """.formatted(  p.id(), p.name(), p.description().trim(), p.coveragesText().trim(), p.audienceText().trim());   TextSegment detailsSeg = TextSegment.*from*(detailsText, detMd);   Embedding detailsEmb = embeddingModel.embed(detailsSeg).content();  store.add(detailsEmb, detailsSeg);  } } |

El método **ingestAll** es el que realiza la carga de productos en la tabla product\_embeddings utilizando la clase de **Langchain4j** llamada EmbeddingStore.

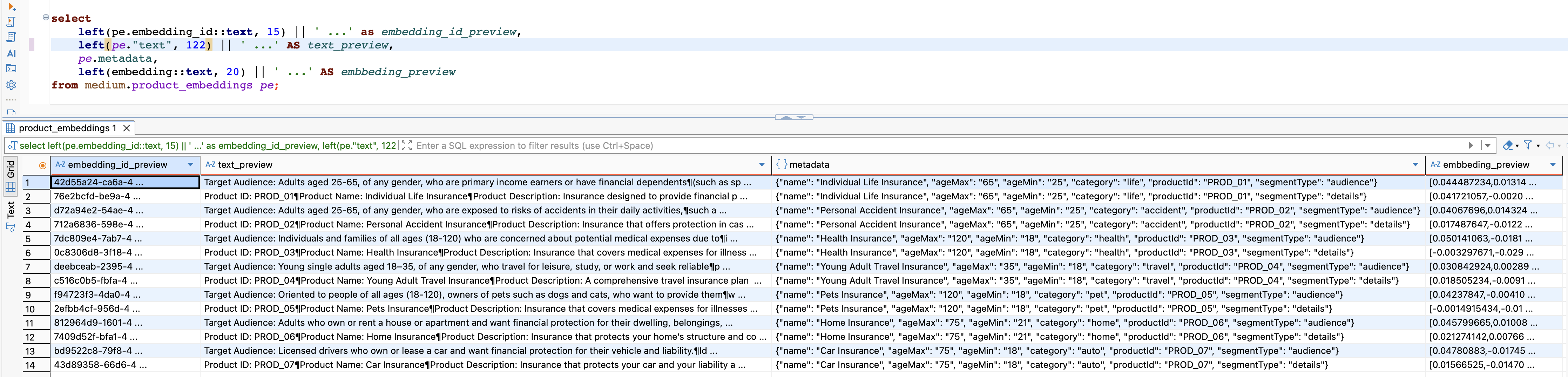
Aquí me gustaría explicar con mayor profundidad este método ya que tiene varias cosas interesantes.

En primer lugar, quiero indicar que en la presente entrega usaremos una estrategia un poco diferente a la utilizada en la Parte 3 de esta serie. Esto es, cargaremos en la tabla product\_embeddings **dos embedings distintos**:

* **Un embedding para el texto que describe la audiencia objetivo**. Esto nos permitirá hacer búsquedas semánticas más precisas en base a la descripción del perfil del cliente, lo cual veremos con más detalle más adelante.
* **Un embedding para la descripción completa del prododucto** (ID, nombre, descripción, coberturas y también su audiencia objetivo). Esto nos permitirá hacer búsquedas semánticas en base al texto completo del producto.
* Por cada uno los embedings cargaremos **metadata**, utilizando el objeto de **Langchain4j** llamado Metadata.

A continuación un ejemplo (resumido) del contenido de la tabla después de la ingestión de los productos:

|  |
| --- |
| **Select**  **left**(pe.embedding\_id::**text**, 15) || ' ...' **as** *embedding\_id\_preview*,  **left**(*pe*."text", 122) || ' ...' **AS** *text\_preview*,  *pe*.metadata,  **left**(embedding::**text**, 20) || ' ...' **AS** *embbeding\_preview*  **from** medium.product\_embeddings *pe*; |



Como veremos en unos párrafos más adelante, la metadata nos será útil para aplicar filtros específicos a nuestra búsqueda de productos, como por ejemplo filtros en función a la **edad del cliente**. Esto lo haremos así ya que los modelos LLM no son muy buenos con los rangos de números (no son muy precisos con los números), por lo cual construiremos este tipo de filtros nosotros mismos utilizando las propiedades ageMax y ageMin que son parte de la metadata cargada. Esto nos dará resultados mucho más robustos y precisos.

En segundo lugar, es necesario destacar que la ingesta de productos en la tabla de embeddings se debe ejecutar una sola vez.

Para hacerlo simple, esto lo manejaremos en nuestra clase Runnable (ver más adelante) mediante una variabe booleana llamada firstRun la cual inicializaremos en true una sola vez, para activar la carga inicial de productos y luego la estableceremos en false para las ejecuciones futuras. Lo mantendremos simple por ahora!

Finalmente comentar que, como podrán observar, hemos utilizado algunas constantes para manejar la Metadata. Éstas constantes las he definido en la siguiente clase utilitaria:

|  |
| --- |
| package com.superchat.services;  // Constants related to product fields. Used during the ingestion process, specifically when mapping // the product metadata. public final class ProductFieldsUtil {  public static final String *META\_PRODUCT\_ID* = "productId";  public static final String *META\_NAME* = "name";  public static final String *META\_AGE\_MIN* = "ageMin";  public static final String *META\_AGE\_MAX* = "ageMax";  public static final String *META\_CATEGORY* = "category";  public static final String *META\_SEGMENT\_TYPE* = "segmentType";  public static final String *SEG\_AUDIENCE* = "audience";  public static final String *SEG\_DETAILS* = "details";   private ProductFieldsUtil() {} } |

Con todo esto ya tenemos una buena idea de cómo ingestar nuestros productos en la tabla de embeddings dentro de una base de datos **PostgreSQL** (para más detalles respecto a la creación del esquema de la tabla de embeddings, consulta por favor la Parte 3 de esta serie).

TODO: explicar que en la parte 3, siembre se envía la BD completa de productos al LLM en cada iteración, lo cual es poco eficiente en cuanto a tiempo de respuesta y costos, sobre todo si tenemos bases de datos grandes de productos. Por eso es mejor aplicar búsquedas semánticas personalizadas y filtros específicos usando la metadata, para luego enviar un conjunto reducido de productos al LLM para que éste lo utilice como contexto.

TODO: incluir un diagrama de arquitectura.