Hanoi University of Science and Technology
School of Information and Communication Technology

Project Report
Object-Oriented Programming
Quick Lap

Topic: Laptop Recommendation System
Integrated Information Retrieval and Large Language Model

Group:	7	
Members:	Nguyen Tuan Duong	20235924
	Nguyen Duc Anh	20235890
	Hoang Quoc Cuong	20235903
	Nguyen Minh Duc	20235915
	Pham Minh Duc	20235918
Supervisor:	TS. Trinh Tuan Dat	

Hanoi, 5/2025

Contribution

Nguyen Tuan Duong - 20235924	Implement RAG; Slide Maker	20%
Nguyen Duc Anh - 20235890	Implement Laptop, Product class; Write Report; Design Use-Case and UML Diagram	20%
Nguyen Minh Duc - 20235915	Implement UI controller, FXML	20%
Pham Minh Duc - 20235918	Implement DataCollector	20%
Hoang Quoc Cuong - 20235903	Implement Database, Design Searching Algorithms	20%

TimeLine

- **March (Approx.): Phase 1 - Ideation, Research & Planning**
  - Proposal development and finalization of project ideas.
  - In-depth research into implementation methodologies, covering:
    - Optimal class structure and design patterns.
    - Strategies for AI integration (e.g., LLMs, embedding services).
    - JavaFX for user interface development.
    - Database selection (e.g., PostgreSQL) and interaction strategies.
    - Understanding and planning for RAG (Retrieval-Augmented Generation) integration, if applicable.
- **Early April (Approx.): Phase 2 - Task Allocation**
  - Detailed breakdown of project tasks.
  - Assignment of responsibilities and tasks to team members.
- **April - Early May (Approx.): Phase 3 - Core Development & Code Integration**
  - Intensive coding and development of individual modules and components.
  - Regular integration of code from different team members.
  - Implementation of core functionalities (UI, database, AI integration, business logic).
- **Mid-May - May 29th: Phase 4 - Refinement, Reporting & Presentation Preparation**
  - Comprehensive code review, debugging, and optimization.
  - Iterative testing and refinement of all application features.
  - Finalization of the project report, ensuring all sections are complete and well-documented.
  - Creation and rehearsal of the project presentation slides.
  - Ensuring all deliverables meet project requirements by the May 29th deadline.

Contents

1 Introduction

In the digital era, E-Commerce platforms have become the primary marketplace for consumers seeking to purchase electronic devices such as smartphones, laptops, tablets, and household appliances.

With the overwhelming number of products available and the complexity of user requirements, helping customers quickly find the most suitable product remains a significant challenge.

This project addresses that challenge by developing a question-answering and product recommendation system for e-commerce platforms.

Focusing on selected product categories including smartphones, laptops, and household electronics such as air conditioners and dishwashers the system leverages multiple data sources and modern information retrieval techniques to assist users in making informed purchasing decisions.

We collected and integrated structured product data, customer reviews, and textual product descriptions from various leading e-commerce websites.

To support efficient and accurate product discovery, we experimented with several approaches:

- From traditional keyword-based search and information processing to advanced AI methods such as Retrieval-Augmented Generation (RAG) using large language models (LLMs).
- By evaluating different LLMs and retrieval pipelines, we aim to identify the most effective combination for enhancing user experience and reducing search time.

The project not only provides a practical solution for e-commerce platforms but also serves as a case study for applying modern information retrieval techniques in real-world scenarios.

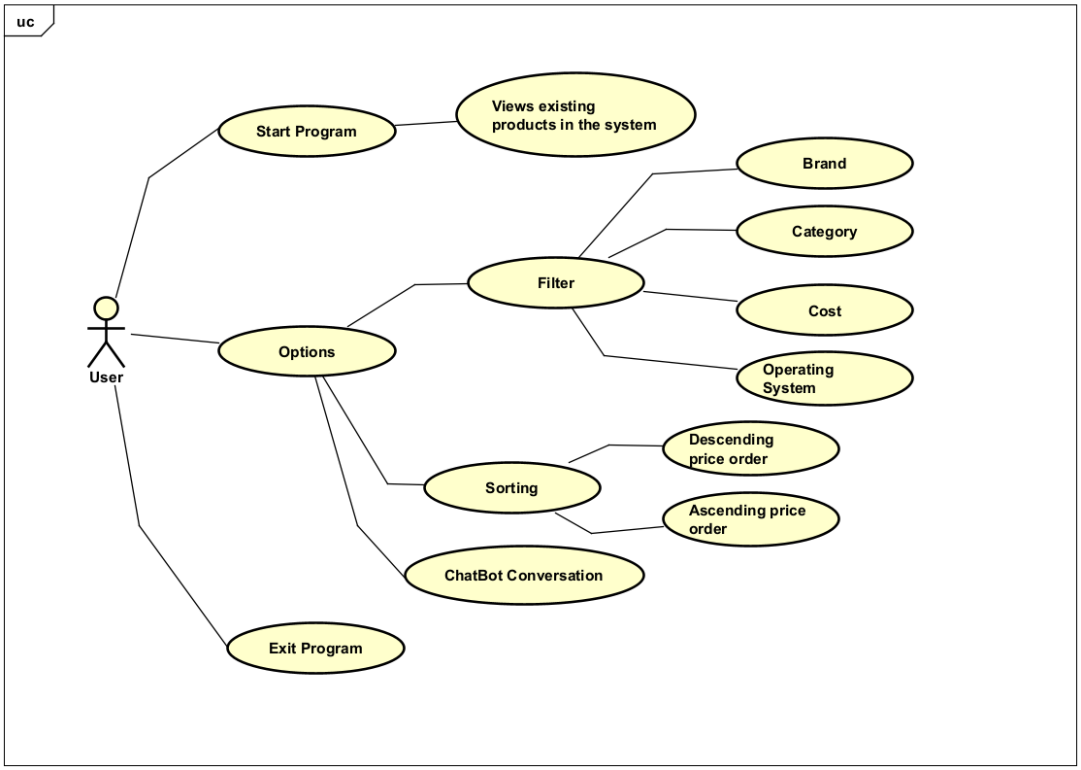
2 System Design

2.1 Requirements

In this project, we aim to develop a Laptop Recommendation System that can:

- Provide users with a list of laptops based on their preferences and requirements.
- Allow users to ask questions about specific laptop models and receive accurate answers.
- Integrate with a database to store and retrieve laptop information.
- Use a Large Language Model (LLM) to enhance the question-answering capabilities.
- The system should be user-friendly, allowing users to easily navigate through the available laptops and ask questions without requiring extensive technical knowledge.

2.2 Use Case Diagram



2.3 Use Case Description

2.3.1 Actors

- **User:** The person interacting with the system to search for and receive recommendations on laptops.

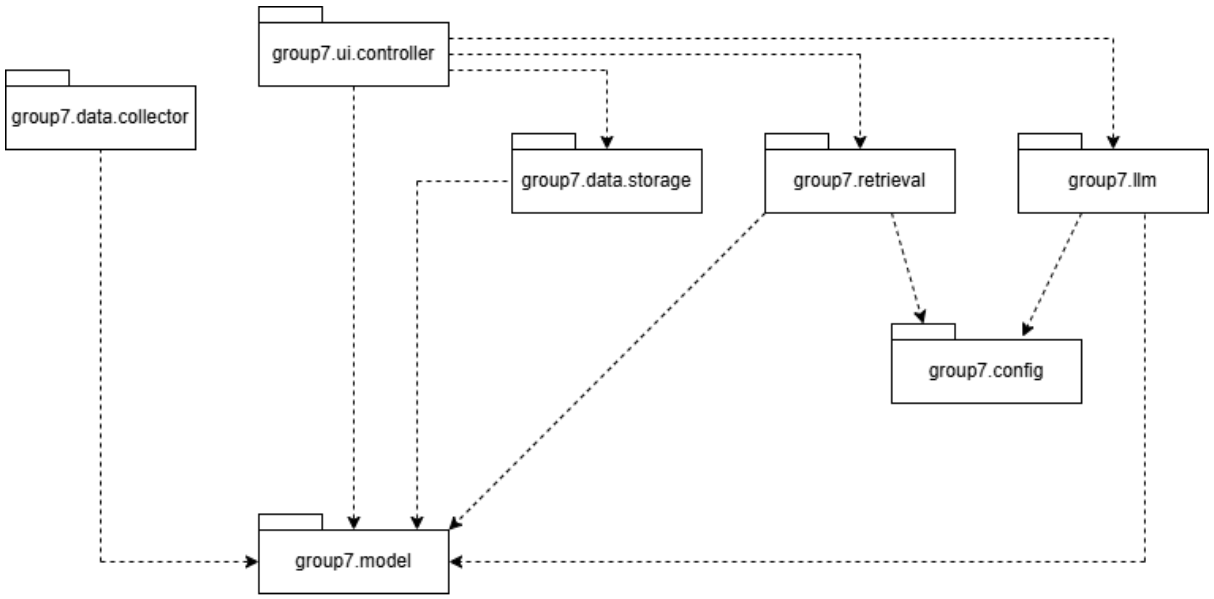
2.3.2 Use Cases

- **Start Program:** The user initiates the system (The entry point for all further interactions).
  - *Views Existing Products in the System:* The user can browse all available laptop products stored in the system.

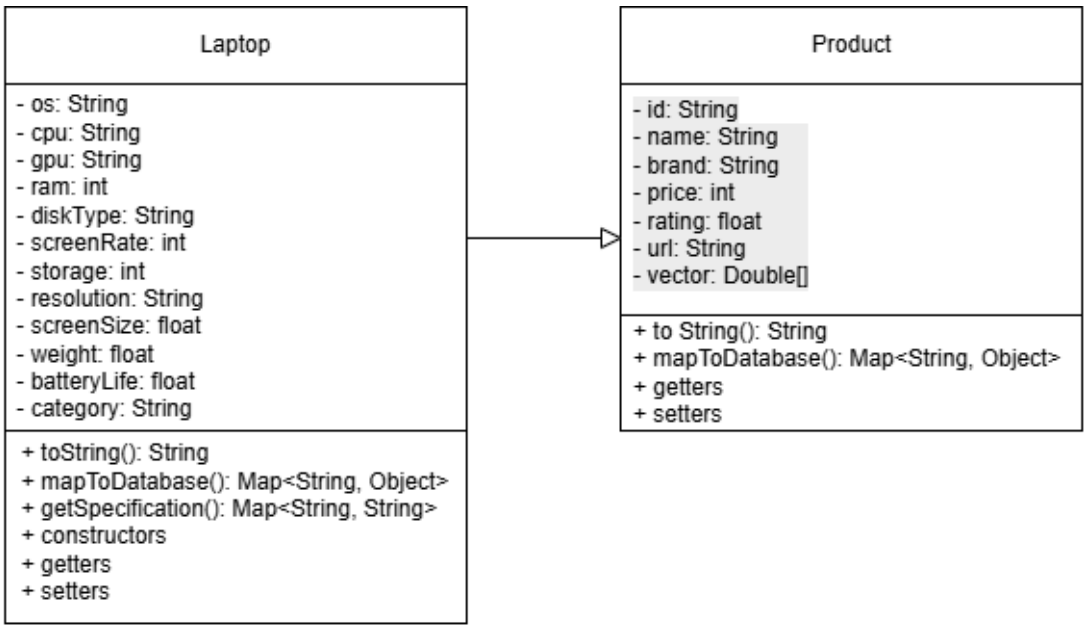
- **Options:** Once the program starts, the user accesses different options that include filtering, sorting, and engaging in chatbot conversation.
  - *Filter:* Allows the user to narrow down the list of products based on specific criteria:
    * **Brand:** Filters laptops by manufacturer (e.g., Dell, HP, Apple).
    * **Category:** Filters by use case or type (e.g., gaming, ultrabook, business).
    * **Cost:** Filters by price range.
    * **Operating System:** Filters based on OS (e.g., Windows, macOS, Linux).
  - *Sorting:* The user can sort the displayed products by price:
    * **Descending Price Order:** Highest price to lowest.
    * **Ascending Price Order:** Lowest price to highest.
  - *ChatBot Conversation:* This use case allows the user to engage with a chatbot powered by a large language model (LLM) to receive personalized recommendations or ask questions in natural language.
- **Exit Program:** Ends the session and closes the system.

3 Architecture

Package Diagram



3.1 Package model (`group7.model`)



The `group7.model` package is responsible for defining the core data structures, or "models," that represent the entities within the application. It establishes a clear hierarchy for product-related data, facilitating organized data management, reusability, and extensibility.

3.1.1 Class `Product`

The `Product` class serves as a base (or parent) class for all types of products in the system. It encapsulates common attributes and functionalities shared across different product categories.

- **Attributes (Fields):**
  - `id: String` - A unique identifier for the product.
  - `name: String` - The display name of the product.
  - `brand: String` - The manufacturer or brand of the product.
  - `price: int` - The price of the product (integer representation).
  - `rating: float` - The customer rating of the product.
  - `url: String` - A URL link to the product's page or image.
  - `vector: double[]` - A numerical vector representation (embedding) of the product, crucial for similarity searches and recommendation features.
- **Methods:**
  - `Product()` - Default constructor.
  - `Product(id, name, brand, price, rating, url)` - Parameterized constructor to initialize common product attributes (note: vector is set separately).
  - `mapToDatabase(): Map<String, Object>` - Likely returns a map representation of the product's data, suitable for database persistence or data transfer.
  - `toString(): String` - Provides a string representation of the product, often used for logging or basic display.
  - Standard getter and setter methods for all its attributes (e.g., `getId()`, `setId(String id)`, `getVector()`, `setVector(double[] vector)`, etc.).

The `Product` class forms the foundation for a polymorphic design, allowing different product types to be treated uniformly where appropriate, while also providing a common interface for accessing shared data.

3.1.2 Class `Laptop`

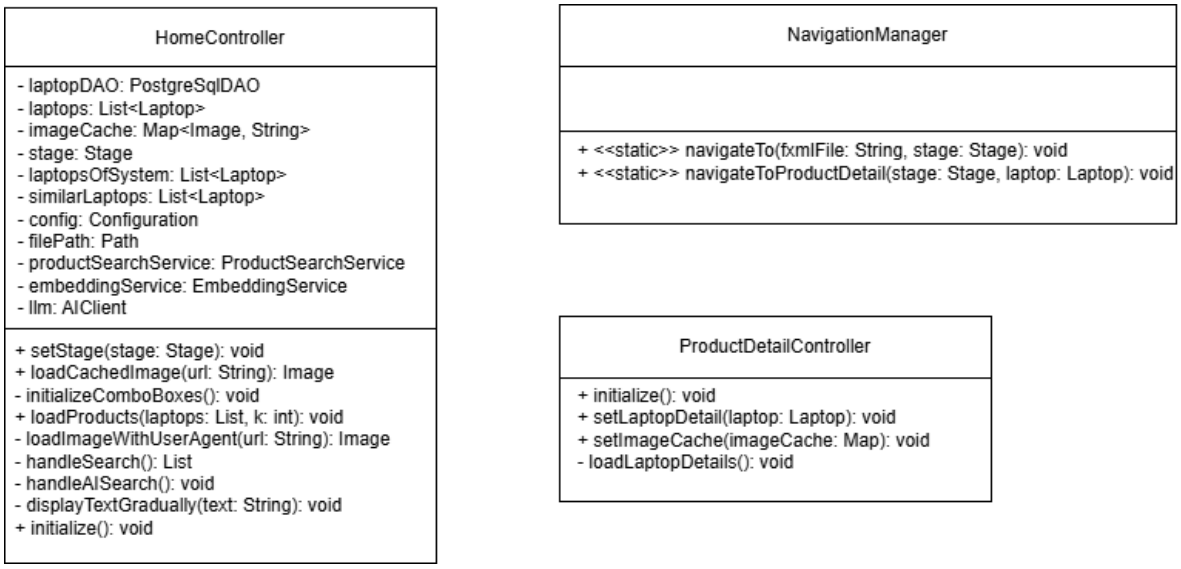
The `Laptop` class is a specialized type of `Product`, inheriting all attributes and methods from the `Product` class and adding specific details relevant to laptops.

- **Inheritance:** `Laptop` extends `Product`.
- **Attributes (Fields) - Specific to Laptop:**
  - `os: String` - Operating System (e.g., "Windows 11", "macOS").
  - `cpu: String` - Central Processing Unit (e.g., "Intel Core i7").
  - `gpu: String` - Graphics Processing Unit (e.g., "NVIDIA GeForce RTX 3060").
  - `ram: int` - Random Access Memory in GB (e.g., 16).
  - `diskType: String` - Type of storage disk (e.g., "SSD", "HDD").
  - `screenRate: int` - Screen refresh rate in Hz (e.g., 120).
  - `storage: int` - Storage capacity in GB or TB (e.g., 512 for 512GB).
  - `resolution: String` - Screen resolution (e.g., "1920x1080").
  - `screenSize: float` - Diagonal screen size in inches (e.g., 15.6).
  - `weight: float` - Weight of the laptop in kg or lbs.
  - `batteryLife: float` - Estimated battery life in hours.
  - `category: String` - Specific laptop category (e.g., "Gaming", "Ultrabook", "Business").
- **Methods:**
  - `Laptop()` - Default constructor.
  - `Laptop(id, name, brand, category, os, price, ..., url)` - A comprehensive parameterized constructor that initializes all attributes from both `Product` and `Laptop`.
  - `getSpecification(): Map<String, String>` - Returns a map of the laptop's detailed technical specifications, likely used for display on product detail pages.
  - `toString(): String` - Overrides the `Product.toString()` method to include laptop-specific information in its string representation.
  - `mapToDatabase(): Map<String, Object>` - Overrides the `Product.mapToDatabase()` method to include laptop-specific data when preparing data for storage.
  - Standard getter and setter methods for all its specific attributes (e.g., `getOs()`, `setOs(String os)`, `getCpu()`, etc.).

OOP Techniques Used:

- The relationship between `Laptop` and `Product` is a classic example of **Inheritance** (an "is-a" relationship), promoting code reuse and allowing for flexible handling of different product types.
- This structure makes it easy to add other product types in the future (e.g., `Smartphone`, `Tablet`) by having them also extend the `Product` class.

3.2 Package ui (User Interface `group7.ui`)



3.2.1 Overview

Create an interactive interface for the user, including components like buttons, search bars, images, etc. This links to corresponding services to provide functionality to the user.

- **Main components of UI code:** Code originating from the `Main.java` file; FXML files for building the interface; Controller classes linked to their respective FXML files.

3.2.2 Process

- Research
- Wireframing & Prototyping
- Testing

3.2.3 UI Design

3.2.4 `HomeController.java` Class

- **Purpose:** This class controls "`Home.fxml`" - the application's main screen. It allows the user to:
  - View a list of laptop products displayed as product cards.
  - Search for laptops by criteria: brand, category, operating system, or keyword.
  - Switch to the chat screen with the chatbot when the chat button is pressed.
- **Function Structure:**
  - `initialize()`: Controller initialization function, called automatically when FXML is loaded.
    Steps to build the function:
    * Initialize DAO (Data Access Object) to connect to the PostgreSQL database.
    * Retrieve all laptop data from the database.
    * Create and display product cards on the `GridPane`.
    * Set up listeners for the search box and `ComboBoxes`.
    * Set up the event for the chat button.
    * Initialize values for `ComboBoxes`.
  - `setStage(Stage stage)`: Sets the current stage (window) for the controller.
    Steps to build the function:
    * Store a reference to the stage to enable navigation between scenes.

- `initializeComboBoxes()`: Initializes values for the product filter `ComboBoxes`.
  Steps to build the function:
  * Add selection values for the brand `ComboBox`.
  * Add selection values for the category `ComboBox`.
  * Add selection values for the operating system `ComboBox`.
  * Set the default value to "All" for all `ComboBoxes`.
- `handleSearch()`: Handles product search and filtering based on criteria.
  Steps to build the function:
  * Retrieve all products from the database.
  * Get filter values from `ComboBoxes` and the search box.
  * Filter products by brand if selected.
  * Filter products by category if selected.
  * Filter products by operating system if selected.
  * Reload the list of filtered products.
- `switchToChatbot(MouseEvent event)`: Handles the event to switch to the chatbot screen.
  Steps to build the function:
  * Currently only prints a console message, can be expanded to switch scenes later.

3.2.5 `NavigationManager.java` Class

- **Purpose:** This class acts as a navigation manager within the application. It allows:
  - Switching between different screens (scenes) in the application.
  - Handling the loading of FXML files and scene switching.
  - Passing data between screens during transitions.
- **Function Structure:**
  - `navigateTo(String fxmlFile, Stage stage)`: Navigates to a new screen specified by the FXML file.
    Steps to build the function:
    * Create an `FXMLLoader` to load the FXML file from the view directory.
    * Create a new `Scene` from the loaded FXML.
    * Set the new `Scene` for the current `Stage`.
    * Display the `Stage` with the new `Scene`.
  - `navigateToProductDetail(Stage stage, String laptopId)`: Navigates to the product detail screen with a specific laptop ID.
    Steps to build the function:
    * Create an `FXMLLoader` to load the `ProductDetail.fxml` file.
    * Create a new `Scene` from the loaded FXML.
    * Get the controller from the `FXMLLoader`.
    * Pass the `laptopId` to the controller of the product detail screen.
    * Set the new `Scene` for the current `Stage`.
    * Display the `Stage` with the new `Scene`.

3.2.6 `ProductDetailController.java` Class

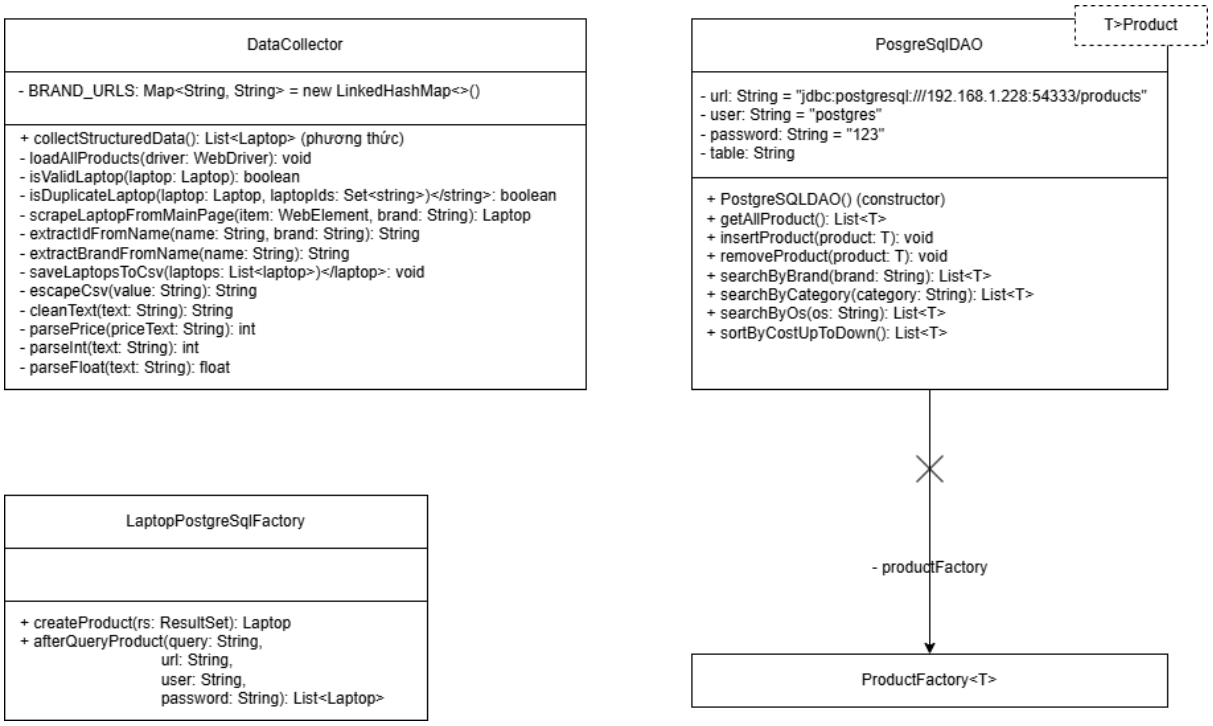
- **Purpose:** This class controls "`ProductDetail.fxml`" - the laptop product detail screen. It allows the user to:
  - Display full detailed information of a specific laptop product.
  - Handle the event to return to the main screen (`Home.fxml`).
  - Load and display product data based on the passed ID.

- **Function Structure:**
  - `initialize()`: Controller initialization function, called automatically when FXML is loaded.
    Steps to build the function:
    * Initialize DAO (Data Access Object) to connect to the PostgreSQL database.
    * Set up the event for the back button to return to the main screen.
  - `setStage(Stage stage)`: Sets the current stage (window) for the controller.
    Steps to build the function:
    * Store a reference to the stage to enable navigation between scenes.
  - `setLaptopId(String laptopId)`: Sets the laptop ID and loads detailed information.
    Steps to build the function:
    * Store the `laptopId` in a member variable.
    * Call the `loadLaptopDetails()` function to load product information.
  - `loadLaptopDetails()`: Loads and displays detailed information of the laptop.
    Steps to build the function:
    * Retrieve all products from the database.
    * Find the product with the ID matching `laptopId`.
    * If found, display all product information on corresponding Labels, format numerical values (price, rating, screen size, etc.), load and display the product image from URL.
    * If not found, display a "Product not found" message.

3.2.7 FXML Files

- **Purpose:** These files define the user interface structure and appearance.
  - "`Chat.fxml`": User interface for the chatbot conversation screen, including an area to display chatbot responses.
  - "`Home.fxml`": User interface for the application's main screen, including: a header with a background image and search bar, a product filter area, and an area to display the product list in a grid.
  - "`ProductCard.fxml`": Interface for a single product card (laptop), including: product image, product name, product price, view details button.
  - "`ProductDetail.fxml`": Interface for the laptop product detail screen, including: product image, basic information (name, rating, price), detailed product description, full technical specifications.
- **Structure:** FXML files use common and diverse UI components such as:
  - *Layout Containers:*
    * `VBox` (vertical layout)
    * `HBox` (horizontal layout)
    * `GridPane` (grid layout)
    * `StackPane` (stacking layers)
  - *Controls:*
    * `Label`: Displays text
    * `Button`: Interactive button
    * `TextField`: Input field
    * `ComboBox`: Dropdown for selecting values
    * `TextArea`: Displays long text
  - *Media:*
    * `ImageView`: Displays images
    * `Image`: Image source
  - *Utilities:*
    * `ScrollPane`: Creates a scrollable frame
    * `FXCollections`: Creates lists for `ComboBoxes`

3.3 Package data (`group7.data`)



3.3.1 Package collector (`group7.data.collector` (`DataCollector.java`))

- **Main Idea:** The `DataCollector` class is designed to collect information about laptop products from an online retail website (https://www.thegioididong.com) by using Selenium WebDriver to automate web browsing, extract product information, and store it in a CSV file. The data is organized into Laptop objects to ensure structure and ease of use in other applications.
- **Class Operation:**
  1. *Initialization and Configuration:*
     - Uses `LinkedHashMap` to store a list of laptop brands and their corresponding URLs (GAMING, AI, MACBOOK, HP, ASUS, DELL, ACER, LENOVO, MSI).
     - Configures `ChromeDriver` in headless mode (no GUI displayed) to optimize performance and automatically downloads the appropriate driver using `WebDriverManager`.
  2. *Data Collection (`collectStructuredData`):*
     - Iterates through each brand and URL in `BRAND_URLS`.
     - Opens the webpage using WebDriver, scrolls the page to load all products (`loadAllProducts`).
     - Finds all product elements (`li.item`) on the page and extracts detailed information using the `scrapeLaptopFromMainPage` method.
     - Checks for validity and duplication of laptops before adding them to the laptops list.
     - Limits the number of laptops collected to a maximum of 360 (`MAX_LAPTOPS`).
  3. *Information Extraction (`scrapeLaptopFromMainPage`):*
     - Retrieves information from HTML elements (name, price, rating, image, RAM, storage capacity, CPU, GPU, screen size, resolution, etc.).
     - Processes raw data using utility methods (`cleanText`, `parsePrice`, `parseInt`, `parseFloat`) to clean and convert formats.
     - Determines the actual brand and category based on the product name or provided brand.
  4. *Data Storage (`saveLaptopsToCsv`):*

- Saves the list of laptops to a CSV file with UTF-8 encoding, adding a BOM to support displaying Vietnamese characters.
- Each laptop is saved as a row with information fields separated by commas, ensuring special characters are handled by the `escapeCsv` method.

5. *Resource Management:*
   - Closes the browser (`driver.quit()`) after data collection to free up resources.

- **Advantages of the class:**
  - *Efficient Automation:* Uses Selenium WebDriver to automate web browsing and scrolling, helping to load all products without manual intervention. Headless mode increases performance and reduces resource usage.
  - *Clear Data Structure:* Data is organized into `Laptop` objects, making it easy to reuse and integrate into other systems. Uses `LinkedHashMap` to maintain brand order and `HashSet` for quick duplicate checking.
  - *Good Error Handling:* Handles exceptions at multiple levels (URL connection, product processing, HTML element access), ensuring the program does not terminate unexpectedly. Checks for validity (`isValidLaptop`) and duplication (`isDuplicateLaptop`) to ensure data quality.
  - *Easy Maintenance and Expansion:* Code is divided into small methods with specific functions, making it easy to modify or add features. URLs and brands are stored in `BRAND_URLS`, easy to update or supplement.
  - *Optimized Performance:* Uses `implicitlyWait` to reduce waiting time when finding elements. Measures the processing time for each product to monitor performance.
- **OOP Techniques Used:**
  - *Encapsulation:* Attributes like `BRAND_URLS` are declared as `private static final`, ensuring they are not modified from outside. Helper methods (`cleanText`, `parsePrice`, etc.) are defined as `private` for internal use within the class.
  - *Abstraction:* The `DataCollector` class provides a simple interface (`collectStructuredData`) to collect data without the user needing to know the internal implementation details (Selenium, scrolling, HTML processing).
  - *Reusability:* Utility methods (`cleanText`, `parsePrice`, `parseInt`, `parseFloat`) can be reused in other classes or projects. The `Laptop` data structure can be integrated into other systems (e.g., data analysis, UI display).

3.3.2 Package storage(`group7.data.storage`)

Interface `ProductFactory<T>`

- Has two functions: the first function returns a product of type T from the query result in the database.
- The second function returns a list of products of type T after passing a query to the database with a URL, user, and password.
- Implemented by the abstract class `SqlFactory()`.

Class `LaptopPostgreSqlFactory` **implements** `ProductFactory<Laptop>`

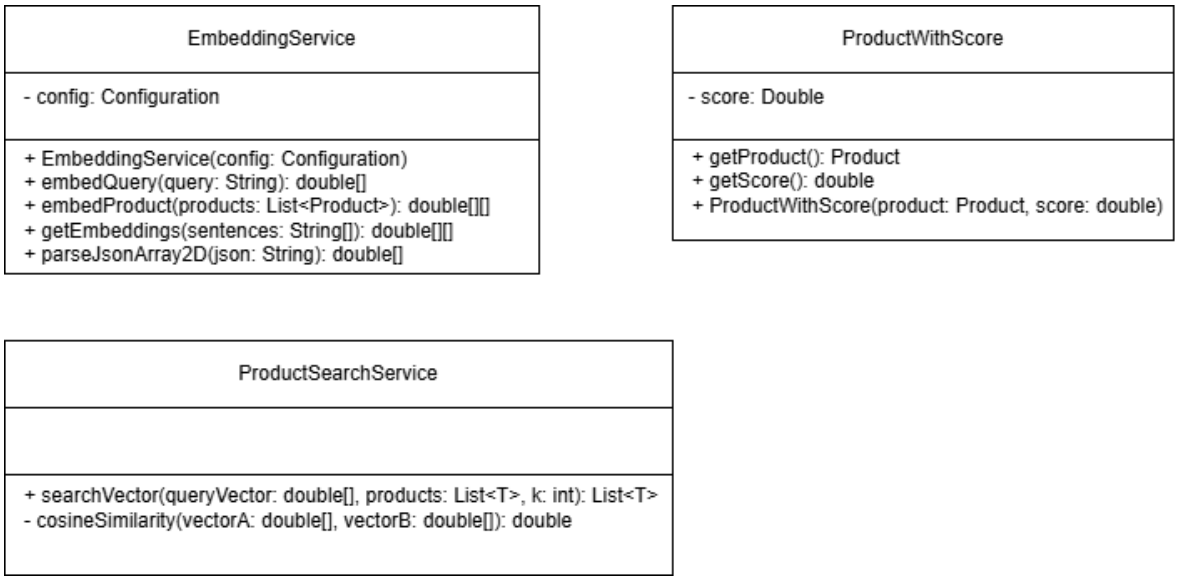
- The `LaptopPostgreSqlFactory` class is used to create laptop objects from a query.
- If other product types or database types need to be added, similar Factory classes inheriting `SqlFactory` would need to be written.
- For example, `PhoneMySqlFactory`.

Class `PostgreSqlDAO<T extends Product>`

- Used to access the database and handle functions as defined by `ProductDAO<>`.

- Uses generic programming to comfortably switch between product types available in the PostgreSQL database.
- If we want to use another database type, we just need to write another DAO class. In `PostgreSqlDAO`, we have an attribute `ProductFactory<T> productFactory`, which helps separate logic: `postgresqlDAO` only needs to handle database connection and queries, while the factory handles data conversion.
- Additionally, it helps with reusability. For instance, if working with laptops, I would assign the `productFactory` attribute as `new LaptopPostgreSqlFactory()`, T := `Laptop`. Or if working with phones, then `productFactory = new PhonePostgreSqlFactory()`.
- In general, when wanting to work with a specific database type and product type, one only needs to care about the type of T, the table name of the product, and the `productFactory`.
- For example, for laptops: `ProductDAO<Laptop> admin = new PostgreSqlDAO<Laptop>("laptop", new LaptopPostgreSqlFactory());`. We have created an admin reference that can perform all functions in the `ProductDAO<>` interface.
- If we want to switch to handling Phones, we only need to change it to `ProductDAO<Phone> admin = new PostgreSqlDAO<Phone>("phone", new PhonePostgreSqlFactory());`.

3.4 Package retrieval (`group7.retrieval`)



Package `group7.retrieval` is designed to handle tasks related to information retrieval in the system, specifically searching for products based on vector embeddings and calculating similarity. The system's recommendation model works by representing products and user queries as numerical vectors and calculating cosine similarity, then providing suggestions to the user.

3.4.1 Class `ProductWithScore`

- The `ProductWithScore` class in package `group7.retrieval` is designed to encapsulate a `Product` object along with its corresponding score, used in search and product ranking tasks.
- This class combines a `Product` object with a score value, typically representing the relevance of the product to a search query.
- The `product` attribute is marked as `final`, ensuring that the `Product` object cannot be changed after initialization, increasing safety and consistency.
- `ProductWithScore` can contain any object belonging to the `Product` class or its subclasses, serving the purpose of reusability for this class in the future when the system expands to products beyond Laptops.

3.4.2 Class `ProductSearchService`

The `ProductSearchService` class in package `group7.retrieval` uses the `cosineSimilarity` function to compare the query vector with product vectors, rank products by similarity, and return a list of the k most suitable products. The `searchVector` method receives the following inputs:

- `queryVector`: Vector representing the search query (usually created by `EmbeddingService`).
- `products`: List of products (of type `T extends Product`) to search.
- `k`: Maximum number of products to return.

This method returns a list of k products with the highest similarity to `queryVector`.
Operating Principle:

- This method creates a list `scoredProducts` (of type `List<ProductWithScore>`) to store products along with their similarity scores. For each product in `products`, its vector is retrieved, and its similarity to `queryVector` is calculated using the `cosineSimilarity` function. A `ProductWithScore` object is created and stored in the `scoredProducts` list for sorting.
- The `scoredProducts` list is sorted by score in descending order (using `sort` with `Double.compare`), then the first k products are taken from the list, cast to type T, and added to the results list to be returned.
- This method uses a generic type (`<T extends Product>`) for generalization. T is a generic type constrained by `extends Product`, meaning T must be a specific class or a subclass of the `Product` class (from package `group7.model`). For example, T can be `Laptop`, `Smartphone`, `Tablet`, or any class that inherits from `Product`.

Benefits:

- *Flexibility:* The method can handle various product types (laptop, smartphone, etc.) without requiring separate code for each type.
- *Separation of search logic and product representation:* The method is not concerned with the specific details of the product (like attributes of `Laptop` or `Smartphone`). It only needs a query vector (`queryVector`) and a list of products with a `getVector()` method that returns an embedding vector.
- *Type safety:* Generic types ensure that the input list (`products`) and the result list (`List<T>`) have the same type T, reducing the risk of unsafe type casting errors at runtime.
- *Code reusability:* A single method can be used for multiple product types, instead of writing separate methods for each.

The `cosineSimilarity` function calculates the cosine similarity between two vectors, returning a value in the range [-1, 1]:

$$\text{cosineSimilarity} = \frac{\text{dotProduct}}{\sqrt{\text{normA}} \cdot \sqrt{\text{normB}}}$$

3.4.3 Class `EmbeddingService`

- The `EmbeddingService` class in package `group7.retrieval` is designed to create vector embeddings for queries and products, using an external API to convert text into numerical vectors. This class can handle any product type belonging to the `Product` class or its subclasses, as long as the product has a `toString()` method to represent it as text.
- `private final Configuration config`: Stores the configuration object (`Configuration`), providing information such as the API embedding URL (`config.getApiUrl()`). Using `final` ensures immutability, increasing safety and consistency.
- The `embedQuery` method converts a text query into an embedding vector. This method provides the `query vector` to `ProductSearchService` for comparison with product vectors.
- The `embedProducts` method converts a list of products into embedding vectors.

- Using `List<? extends Product>` instead of `List<Product>` allows handling any list of products belonging to the `Product` class or its subclasses (like `Laptop`, `Smartphone`, etc.).
- This makes the method flexible, applicable to various product types without code changes.
- The `getEmbeddings` method sends an HTTP POST request to the API to get embeddings for a list of sentences.

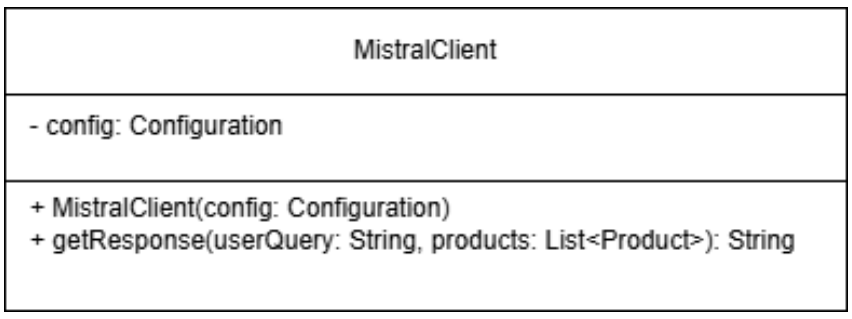
How it works:

1. *Build JSON:*
   - Create a JSON string with the format {`"sentences": ["sentence1", "sentence2", ...]`}.
   - Escape """ characters in sentences to ensure valid JSON.
2. *Send HTTP request:*
   - Open a connection to the URL from `config.getApiUrl()`.
   - Set the method to POST, and headers `Content-Type` and `Accept` to `application/json`.
   - Send the JSON string via `OutputStream`.
3. *Receive and process response:*
   - Read the response from `InputStream` and save it to a `StringBuilder`.
   - Call `parseJsonArray2D` to convert JSON into a `double[][]` array.
4. *Disconnect:* Close the HTTP connection after completion.

The `parseJsonArray2D` method uses the Gson library to parse JSON into a `double[][]` array. The `EmbeddingService` class is designed with high generalization, demonstrated by:

- *Support for multiple product types.* The `embedProducts` method uses `List<? extends Product>`, allowing it to process any list of products that are subtypes of `Product`. This ensures the class can work with any product type, as long as they have a valid `toString()` method.
- *Configuration separation.* Using `Configuration` to get `getApiUrl()` helps the class not depend on a specific URL. This allows easy API or configuration changes without modifying the code.
- *Independence from product details.* The class only requires products to provide text via `toString()`. This makes `EmbeddingService` unaware of the detailed structure of `Product` classes or their subclasses, increasing abstraction.
- *Batch processing support:* The `getEmbeddings` method accepts a `String[]` array, allowing embeddings for multiple sentences (queries or products) to be created simultaneously, optimizing communication with the API.

3.5 Package llm (`group7.llm`)



Package `group7.llm` is designed to integrate and use Large Language Models (LLMs) to provide responses based on artificial intelligence, especially in the context of product consultation.

Class `MistralClient`

- This class implements the `AIClient` interface and performs specific tasks to interact with the Mistral AI API.
- **Constructor:** Receives a `Configuration` object to get configuration information (API URL, API key), ensuring flexibility and avoiding hard-coding.
- **Method `getResponse`:** Processes the user query and product list, calls the Mistral API to generate a consultation response.

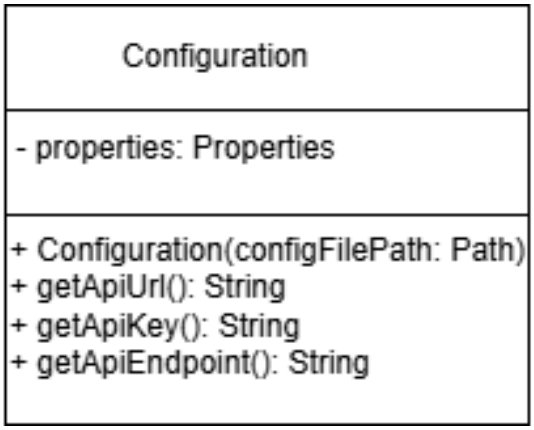
How it works:

- *Build prompt:* Converts the product list into a text string by calling `toString()` on each product. Creates a prompt including:
  - User query (`userQuery`).
  - Product list.
  - Request for LLM to suggest the 5 most suitable products and explain why.
- *Create JSON request:* Uses the `org.json` library to build JSON with:
  - A system message defining the LLM's role as a "useful product consultant".
  - A user message containing the prompt.
  - Parameters like model (Mistral AI model) and temperature (creativity level, default 0.7).
- *Send HTTP request:* Uses `HttpClient` to send a POST request to the Mistral API with:
  - URL from `config.getApiEndpoint()`.
  - Authorization header with API key from `config.getApiKey()`.
  - Body as JSON string.
- *Process response:* Parses the JSON response to get content from the `choices[0].message.content` field. Returns the response content as a string.
- *Error handling:* Catches exceptions and returns a default error message: "Error calling Mistral API".

Generalization Features:

- *Support for multiple product types:* Uses `List<? extends Product>` to allow processing any list of products belonging to the `Product` class or its subclasses.
- *Flexible integration:* Relies on `Configuration` to get URL and API key, making it easy to switch to another API or change configuration.
- *Extensibility:* Since it implements the `AIClient` interface, `MistralClient` can be replaced by other implementations without affecting the calling code.

3.6 Package config (`group7.config`)



Package `group7.config` is responsible for managing application settings, such as API credentials and service URLs. Its main goal is to abstract configuration details away from the core application logic, allowing for easy updates and maintenance without modifying the source code.

Class `Configuration`

- This class is designed to load and provide access to configuration properties from an external file.
- **Constructor:** Takes a `Path` object pointing to the configuration file (e.g., config.properties) and initializes the class by loading the properties from this file.
- **Methods `getApiUrl`, `getApiKey`, `getApiEndpoint`:** These methods provide access to specific configuration values (API URL, key, and endpoint) stored in the properties file. They retrieve the corresponding value for a predefined key (e.g., api.url, api.key).

How it works:

- *Initialization:* An instance of Configuration is created by passing the path to a .properties file.
- *Load Properties:* The constructor uses a FileInputStream to open the specified file. It then initializes a java.util.Properties object and calls its load() method, which parses the input stream from the file. This populates the internal properties object with key-value pairs defined in the file.
- *Data Retrieval:* When a getter method like getApiKey() is called, it uses the getProperty(String key) method of the Properties object to look up and return the value associated with the corresponding key (e.g., "api.key").
- *Error Handling:* The class includes error handling to manage IOException (e.g., if the configuration file is not found or cannot be read), ensuring the application fails gracefully or reports a clear error.

Generalization Features:

- *Separation of Concerns:* By isolating configuration in a separate class and file, the application's business logic (like in MistralClient) is decoupled from its configuration details. You can change the API key or URL without recompiling the code.
- *Environment-Specific Configurations:* This design allows for easy management of different configurations for different environments (development, testing, production). A different .properties file can be supplied at runtime without any code changes.
- *Easy to Extend:* To support new configuration parameters, one only needs to add a new key-value pair to the .properties file and a corresponding getter method in the Configuration class.

4 Technology Stack

4.1 Frameworks & Libraries

1. **Language:**
   - **Java:** The primary programming language, evident from file names (`.java`), class structures, and mentioned libraries/frameworks.
2. **User Interface (UI):**
   - **JavaFX:** Used for building the graphical user interface. Mentions include:
     - FXML: For defining the UI structure declaratively.
     - `Stage`, `Scene`: Core JavaFX windowing components.
     - `ChatController`, `HomeController`, etc.: JavaFX controller classes.
     - JavaFX UI Controls: `Button`, `Label`, `TextField`, `TextArea`, `ComboBox`, `ImageView`, `GridPane`, `VBox`, `HBox`, `StackPane`, `ScrollPane`.
     - JavaFX Animation: `Timeline`, `KeyFrame` (for the typewriter effect).
3. **Data Collection & Web Interaction:**

- **Selenium WebDriver:** For automating web browser interaction to scrape laptop data from "thegioididong.com".
- **ChromeDriver:** Specific WebDriver implementation for Google Chrome.
- **WebDriverManager:** For automatically managing browser driver binaries.

4. **Database & Data Persistence:**
   - **PostgreSQL:** The relational database management system used for storing product data.
   - **JDBC (Implicit):** Likely used for Java-to-PostgreSQL connectivity via the DAO layer, though not explicitly named, it's the standard.
   - **CSV (Comma-Separated Values):** Used as a file format for storing the scraped laptop data (`saveLaptopsToCsv`).
5. **AI & Machine Learning Integration:**
   - **External Embedding API:** An unspecified external API is used to convert text (product descriptions, user queries) into numerical vector embeddings (via `EmbeddingService`).
   - **Mistral AI API:** An LLM (Large Language Model) API used for the chatbot functionality to generate responses to user queries (`MistralClient` interacts with this).
   - **JSON:** Used as the data interchange format for communicating with both the Embedding API and the Mistral AI API.
     - **Gson library:** Explicitly mentioned for parsing JSON responses from the Embedding API.
     - **org.json library:** Explicitly mentioned for constructing JSON requests for the Mistral AI API.
6. **HTTP Communication:**
   - **Java's built-in HTTP Client:** (`java.net.HttpURLConnection` or `java.net.http.HttpClient`) Used by `EmbeddingService` and `MistralClient` to make POST requests to external APIs.
7. **Configuration:**
   - `.properties` **files:** Used for application configuration (e.g., `application.properties` storing API URLs, keys, database credentials).
8. **Design Patterns & Architecture:**
   - **Model-View-Controller (MVC) variant:** The separation of FXML (View), Controller classes, and Model classes (`Product`, `Laptop`) strongly suggests an MVC-like architecture.
   - **Data Access Object (DAO):** `ProductDAO`, `PostgreSqlDAO` are used to abstract and encapsulate database interactions.
   - **Factory Pattern / Abstract Factory Pattern:** `ProductFactory`, `SqlFactory`, `LaptopPostgreSqlFactory` are used for creating product objects, decoupling client code from concrete class instantiation.
   - **Service Layer:** Implied by classes like `EmbeddingService`, `ProductSearchService`, `AlClient`, which encapsulate specific business logic or external service interactions.
9. **Core Java Features:**
   - **Java Collections Framework:** `List`, `Map`, `HashSet`, `LinkedHashMap` are used extensively for data management.
   - **Generics:** Used heavily in DAO and Factory patterns (`<T extends Product>`) for type safety and reusability.
   - **Object-Oriented Programming (OOP):** Encapsulation, Inheritance (`Laptop` extends `Product`), Polymorphism are fundamental to the design.

4.2 Algorithms & Techniques

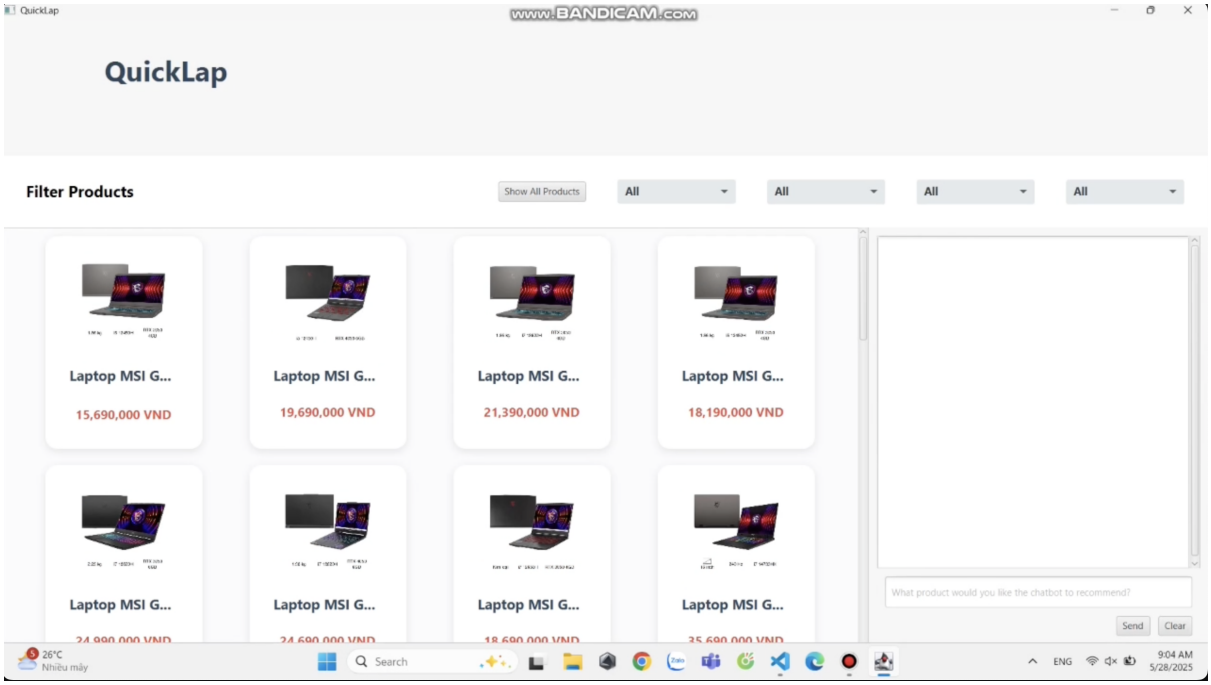
1. **Information Retrieval & Semantic Search:**
   - **Vector Embeddings:** Textual descriptions of products and user queries are converted into dense vector representations. This allows for semantic understanding beyond simple keyword matching.
   - **Cosine Similarity:** The mathematical measure used to determine the similarity between the query vector and product vectors. This is the core of the semantic search functionality (`ProductSearchService.cosineSimilarity`).

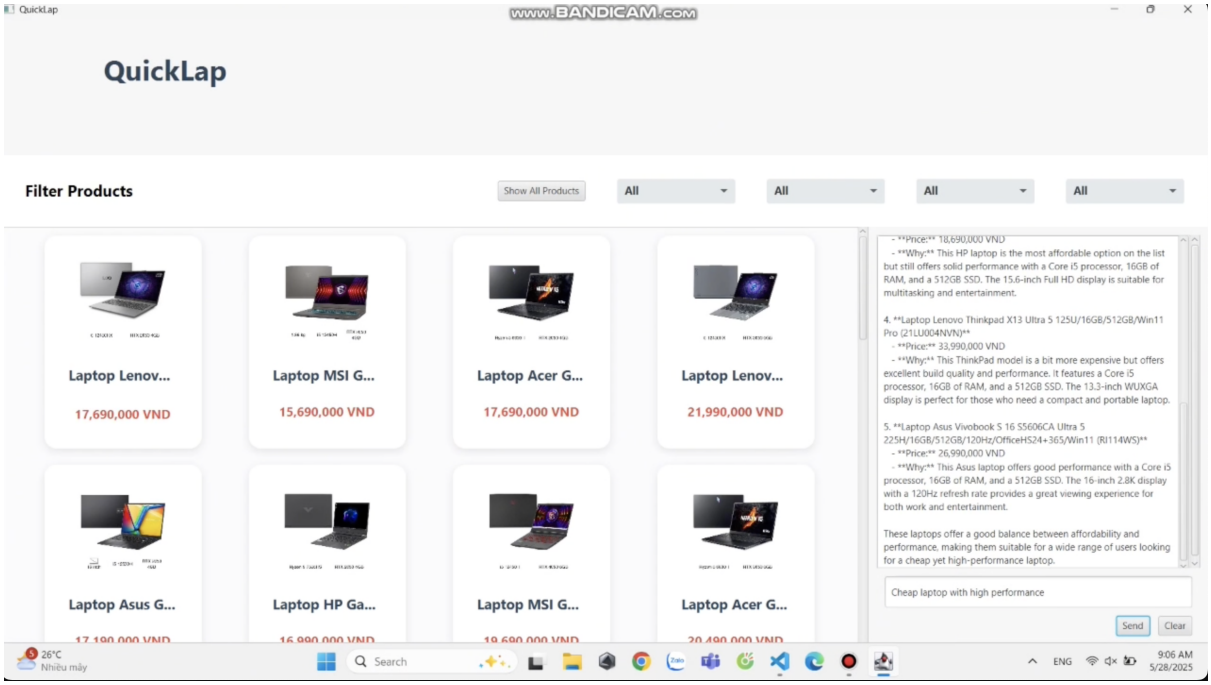
$$\text{cosineSimilarity} = \frac{\vec{A} \cdot \vec{B}}{||\vec{A}|| \cdot ||\vec{B}||}$$

   - **Top-K Ranking/Retrieval:** After calculating similarities, the system ranks products and returns the top k most relevant ones to the user's query.
2. **AI Chatbot Interaction:**
   - **Prompt Engineering:** The `MistralClient` constructs specific prompts (including user query, relevant product context, and instructions) to guide the LLM in generating useful and relevant responses.
   - **LLM API Integration:** Sending structured requests to the Mistral AI API and parsing its JSON responses to extract the generated text.
3. **Web Scraping & Data Extraction:**
   - **Automated Browser Navigation:** Selenium is used to navigate web pages, scroll to load dynamic content (`loadAllProducts`).
   - **HTML Element Selection & Parsing:** Identifying and extracting data from specific HTML elements (e.g., `li.item`, product name, price, specs).
   - **Data Cleaning & Transformation:** Raw scraped data is processed (`cleanText`, `parsePrice`, `parseInt`, `parseFloat`) to ensure consistency and correct data types.
4. **User Interface Logic:**
   - **Gradual Text Display (Typewriter Effect):** Implemented in `ChatController.displayTextGradually` using JavaFX `Timeline` and `KeyFrame` to enhance the user experience when displaying chatbot messages.
   - **Client-Side Filtering:** The `HomeController` implements logic to filter the displayed product list based on user selections in `ComboBoxes` (brand, category, OS) and keyword search.
5. **Data Management & Persistence:**
   - **Object-Relational Mapping (Conceptual):** The DAO layer and methods like `mapToDatabase()` in model classes facilitate the translation of Java objects to a format suitable for storage in the PostgreSQL relational database.
   - **CSV Data Serialization:** Converting lists of `Laptop` objects into a CSV file format for storage/export.

5 Instruction and Demo

Some demo about the project:





6 Conclusion

This project successfully developed a comprehensive Java-based desktop application designed to enhance the user's experience in discovering and learning about laptop products. By integrating a robust data collection pipeline, an intuitive JavaFX user interface, and advanced AI-driven features, the application provides a powerful tool for both casual browsing and specific product inquiry.

Achievements:

- **Automated Data Acquisition:** The successful implementation of a Selenium-based web scraper demonstrates an effective method for gathering up-to-date product information, which is then structured and persisted in a PostgreSQL database. This forms a reliable foundation for the application's data needs.
- **Intelligent Information Retrieval:** The integration of vector embeddings and cosine similarity for product search allows users to find relevant laptops based on semantic understanding, moving beyond simple keyword matching. This significantly improves the quality and relevance of search results.
- **AI-Powered Conversational Assistance:** The incorporation of the Mistral AI LLM via an API provides users with an interactive chatbot capable of answering product-related questions and offering recommendations, making the information discovery process more engaging and efficient.
- **Modular and Maintainable Architecture:** The project's adherence to design patterns such as DAO, Factory, and a clear separation of concerns (UI, business logic, data access) results in a well-structured, maintainable, and extensible codebase. The use of JavaFX for the UI ensures a modern and responsive user experience.
- **Effective Technology Integration:** The project successfully brought together diverse technologies, including Java, JavaFX, Selenium, PostgreSQL, and external AI APIs, showcasing the ability to build complex systems by leveraging appropriate tools for each component.

While the application achieves its primary objectives, potential avenues for future development include expanding the product categories beyond laptops, developing a web-based version for broader accessibility, implementing user accounts for personalized recommendations, and further refining the AI's conversational abilities and analytical depth.

The reliance on external website structures for scraping and external APIs for AI functionalities also presents considerations for long-term maintenance and operational costs.

In conclusion, this project serves as a strong proof-of-concept for an intelligent product consultation system. It effectively demonstrates the power of combining automated data gathering, sophisticated search algorithms, and conversational AI to create a valuable tool for consumers.

The established architecture and technological foundation provide a solid platform for future enhancements and the development of even more sophisticated product discovery and advisory services.