**BỘ KHOA HỌC VÀ CÔNG NGHỆ**

**HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG**

-----------------------------



**BÁO CÁO   
ĐỒ ÁN MÔN HỌC**

**ĐỀ TÀI: XÂY DỰNG ỨNG DỤNG TƯƠNG TRANH   
MÔ PHỎNG XỬ LÝ HÀNG TỒN KHO   
CHO NHẬP XUẤT HÀNG HÓA**

**Môn học:** Các hệ thống phân tán

**Giảng viên hướng dẫn:** Ths. Lê Ngọc Bảo

**Thực hiện bởi nhóm sinh viên, bao gồm:**

1. Ngô Tấn Sang N22DCCN167 <Trưởng nhóm>
2. Văn Minh Tấn N22DCCN175 <Thành viên>
3. Triệu Việt Thành N22DCCN177 <Thành viên>
4. Huỳnh Phát Tài N22DCCN171 <Thành viên>
5. Phạm Tấn Nguyên N22DCCN156 <Thành viên>

**TP.HCM, tháng 12/2025**

# MỤC LỤC

[MỤC LỤC 2](#_Toc214452954)

[DANH SÁCH HÌNH 4](#_Toc214452955)

[TÓM TẮT 6](#_Toc214452956)

[CHƯƠNG I. TỔNG QUAN 7](#_Toc214452957)

[1. Giới thiệu đề tài. 7](#_Toc214452958)

[2. Cơ sở lý thuyết. 7](#_Toc214452959)

[2.1. Kiến trúc Client-Server Phân tán 7](#_Toc214452960)

[2.2. Cơ sở lý thuyết về JavaFX 8](#_Toc214452961)

[2.2.1. Kiến trúc Cốt lõi: Scene Graph (Đồ thị Cảnh) 8](#_Toc214452962)

[2.2.2. Vòng đời Ứng dụng (Application Lifecycle) 8](#_Toc214452963)

[2.2.3. Tách biệt Giao diện và Logic (FXML, Controller, và CSS) 9](#_Toc214452964)

[2.2.4. Khả năng Tương tác và Bố trí (Controls and Layouts) 9](#_Toc214452965)

[2.3 gRPC: 10](#_Toc214452966)

[2.4. Xử lý Tương Tranh (Concurrency Control) 11](#_Toc214452967)

[2.4.1 Các vấn đề khi không xử lý tương tranh: 12](#_Toc214452968)

[2.4.1 Cơ chế đồng bộ hóa trong Lập trình đa luồng: 13](#_Toc214452969)

[1. Mutex (Mutual Exclusion) 13](#_Toc214452970)

[2. Read-Write Lock (Khóa Đọc-Ghi) 15](#_Toc214452971)

[2.5. Inversion of Control (IoC) và Dependency Injection (DI) 16](#_Toc214452972)

[2.5.1. Khái niệm Inversion of Control (IoC): 16](#_Toc214452973)

[2.5.2. Dependency Injection (DI): 17](#_Toc214452974)

[2.5.3. Lợi ích của IoC và DI trong Hệ thống Phân tán 18](#_Toc214452975)

[CHƯƠNG II. THIẾT KẾ HỆ THỐNG 20](#_Toc214452976)

[1. Kiến trúc Tổng quan : 20](#_Toc214452977)

[2. Phân tích Tác nhân và Phân quyền : 21](#_Toc214452978)

[3. Thiết kế Giao tiếp Dịch vụ (gRPC Service Contract) : 22](#_Toc214452979)

[4. Thiết kế Lưu trữ Dữ liệu (Data Storage Design) : 23](#_Toc214452980)

[5. Thách thức Cốt lõi: Rủi ro Tương tranh với File JSON 24](#_Toc214452981)

[CHƯƠNG III. TRIỂN KHAI HỆ THỐNG 26](#_Toc214452982)

[1. Kiến trúc Lớp (Layered Architecture) phía Server 26](#_Toc214452983)

[2. Luồng Xử lý Request Chi tiết (Request Flow with Interceptors) : 29](#_Toc214452984)

[3. Giải pháp Tương tranh và Quản lý Khóa 31](#_Toc214452985)

[3.1. Phân tích các chiến lược đồng bộ hóa 31](#_Toc214452986)

[3.2. Giải pháp đề xuất: Global ReadWriteLock 32](#_Toc214452987)

[4. Giao diện 33](#_Toc214452988)

[4.1. Trang Đăng nhập 33](#_Toc214452989)

[4.2. Trang Tổng quan 34](#_Toc214452990)

[4.3. Trang Quản lý Kho 36](#_Toc214452991)

[4.4. Trang Quản lý Sản phẩm 36](#_Toc214452992)

[4.5. Trang Quản lý Người dùng 38](#_Toc214452993)

[4.6. Trang Lịch sử Giao dịch 40](#_Toc214452994)

[4.7. Trang Báo cáo 41](#_Toc214452995)

[4.8. Trang Hồ sơ Cá nhân 42](#_Toc214452996)

[4.9. Thông báo Hết hạn Phiên 43](#_Toc214452997)

[CHƯƠNG IV. KẾT LUẬN 45](#_Toc214452998)

[1. Kết quả đạt được 45](#_Toc214452999)

[2. Hạn chế và Tồn tại 45](#_Toc214453000)

[3. Hướng phát triển Tương lai 46](#_Toc214453001)

[TÀI LIỆU THAM KHẢO 47](#_Toc214453002)

# DANH SÁCH HÌNH

[Hình 1 - Hình ảnh mô tả cách hoạt động của gRPC 11](#_Toc214452874)

[Hình 2 - Các tác hại nếu không xử lý tương tranh 13](#_Toc214452875)

[Hình 3 - Sơ đồ tuần tự thể hiện cơ chế Mutex 14](#_Toc214452876)

[Hình 4 - Hình ảnh minh họa mutex 14](#_Toc214452877)

[Hình 5 - Hình ảnh minh họa Read-Write Lock 15](#_Toc214452878)

[Hình 6 - Hình ảnh mô tả luồng hoạt động của Read-Write Lock 15](#_Toc214452879)

[Hình 7 - Minh họa phương pháp truyền thống và IoC/DI 18](#_Toc214452880)

[Hình 8 - Sơ đồ các thành phần của hệ thống 20](#_Toc214452881)

[Hình 9 - Sơ đồ usecase tổng quát các chức năng của 2 tác nhân Staff và Manager. 22](#_Toc214452882)

[Hình 10 - Cấu trúc dữ liệu cho entity User 23](#_Toc214452883)

[Hình 11 - Cấu trúc dữ liệu cho entity Product 24](#_Toc214452884)

[Hình 12 - Cấu trúc dữ liệu cho entity Transaction 24](#_Toc214452885)

[Hình 13 - Sơ đồ tuần tự thể hiện việc mất dữ liệu khi không xử lý tương tranh 25](#_Toc214452886)

[Hình 14 - Cấu trúc thư mục Server 26](#_Toc214452887)

[Hình 15 - Kiến trúc tổng quát server. 28](#_Toc214452888)

[Hình 16 - Luồng xử lý Request, Xác thực và Xử lý lỗi 29](#_Toc214452889)

[Hình 17 - Luồng hoạt động về xuất sản phẩm và có mô tả về cách xử lý tương tranh. 30](#_Toc214452890)

[Hình 18 - Sơ đồ tuần tự mô tả 2 giao dịch diễn ra cùng lúc và cách khóa hoạt động 33](#_Toc214452891)

[Hình 19 - Giao diện Trang Đăng nhập 34](#_Toc214452892)

[Hình 20 - Giao diện Dashboard của Manager 35](#_Toc214452893)

[Hình 21 - Giao diện Dashboard của Staff 35](#_Toc214452894)

[Hình 22 - Giao diện Quản lý Kho hàng 36](#_Toc214452895)

[Hình 23 - Giao diện trang Quản lý Sản phẩm 37](#_Toc214452896)

[Hình 24 - Giao diện hộp thoại Thêm Sản phẩm mới 38](#_Toc214452897)

[Hình 25 - Giao diện hộp thoại Chỉnh sửa Sản phẩm 38](#_Toc214452898)

[Hình 26 - Giao diện Quản lý người dùng 39](#_Toc214452899)

[Hình 27 - Giao diện hộp thoại Thêm người dùng mới 39](#_Toc214452900)

[Hình 28 - Giao diện Lịch sử Giao dịch của Manager 40](#_Toc214452901)

[Hình 29 - Giao diện Lịch sử Giao dịch của Staff 41](#_Toc214452902)

[Hình 30 - Giao diện Báo cáo - 1 41](#_Toc214452903)

[Hình 31 - Giao diện Báo cáo - 2 42](#_Toc214452904)

[Hình 32 - Giao diện Hồ sơ cá nhân 43](#_Toc214452905)

[Hình 33 - Giao diện Chỉnh sửa Thông tin cá nhân 43](#_Toc214452906)

[Hình 34 - Giao diện Thông báo hết hạn phiên đăng nhập 44](#_Toc214452907)

# TÓM TẮT

Trong môi trường kinh doanh hiện đại, việc quản lý kho hàng hiệu quả đóng vai trò tối quan trọng. Đồ án này trình bày việc "Xây dựng Ứng dụng tương tranh mô phỏng xử lý hàng tồn kho cho nhập xuất hàng hóa" , một Hệ thống Quản lý Kho phân tán (WMS).

Mục tiêu cốt lõi là xây dựng một hệ thống theo mô hình Client-Server phân tán. Phía Server được phát triển bằng Java 17+ , áp dụng Kiến trúc Lớp (Layered Architecture) nghiêm ngặt và các nguyên lý Inversion of Control (IoC) / Dependency Injection (DI) để đảm bảo tính dễ bảo trì. Kênh giao tiếp giữa Client và Server sử dụng gRPC và Protocol Buffers để đạt hiệu suất cao. Phía Client là một ứng dụng giao diện trực quan, được xây dựng bằng JavaFX, tách biệt rõ ràng giữa giao diện (FXML) và logic (Controller) . Hệ thống cũng triển khai cơ chế xác thực mạnh mẽ bằng JWT và BCrypt, kết hợp với Interceptor để thực thi Kiểm soát Truy cập Dựa trên Vai trò (RBAC) cho hai tác nhân là "Manager" và "Staff" .

Một thách thức thiết kế cốt lõi của dự án là quyết định sử dụng tệp JSON làm nơi lưu trữ dữ liệu, nhằm mục đích làm nổi bật và giải quyết vấn đề tương tranh. Báo cáo phân tích rõ rủi ro "Mất Cập nhật" (Lost Update) khi nhiều luồng cùng ghi tệp . Giải pháp được đưa ra là triển khai thành công cơ chế khóa **ReadWriteLock** ở cấp độ ứng dụng (tại lớp DataSource). Giải pháp này giải quyết triệt để vấn đề toàn vẹn dữ liệu, đồng thời tối ưu hóa hiệu năng cho kịch bản "Đọc nhiều, Ghi ít" của hệ thống.

Kết quả, dự án đã mô phỏng thành công các nghiệp vụ kho cốt lõi (quản lý sản phẩm, nhập/xuất kho, báo cáo) và là một minh chứng thực tế cho việc áp dụng các nguyên lý hệ thống phân tán và xử lý tương tranh vào một ứng dụng nghiệp vụ.

# CHƯƠNG I. TỔNG QUAN

## 1. Giới thiệu đề tài.

Trong môi trường kinh doanh hiện đại, việc quản lý kho hàng và chuỗi cung ứng đóng vai trò tối quan trọng, ảnh hưởng trực tiếp đến hiệu quả hoạt động và khả năng sinh lời của doanh nghiệp. Sự phức tạp ngày càng tăng trong việc theo dõi hàng tồn kho, kiểm soát dòng chảy của sản phẩm, và xử lý chính xác các giao dịch nhập xuất hàng hóa đòi hỏi một giải pháp quản lý mạnh mẽ và đáng tin cậy.

Đề tài mà nhóm thực hiện là **Xây dựng Ứng dụng tương tranh mô phỏng xử lý hàng tồn kho cho nhập xuất hàng hóa**. Đây là một **Hệ thống Quản lý Kho phân tán (Warehouse Management System - WMS)** được phát triển như đồ án môn học cho môn *Các hệ thống phân tán*.

Mục tiêu cốt lõi của dự án là xây dựng một hệ thống hoạt động theo mô hình Client-Server phân tán, tập trung vào các điểm sau:

* **Tính Phân tán và Hiệu suất:** Hệ thống được xây dựng bằng ngôn ngữ **Java 17+** và sử dụng **gRPC** (Remote Procedure Call) với Protocol Buffers để thiết lập một kênh giao tiếp hiệu suất cao, tin cậy và hiệu quả giữa máy khách (Client) và máy chủ (Server).
* **Xử lý Nghiệp vụ Kho:** Mô phỏng và quản lý các nghiệp vụ cốt lõi của kho hàng, bao gồm:
  + Quản lý danh sách sản phẩm (Product Management).
  + Thực hiện các giao dịch **nhập hàng** (tăng tồn kho) và **xuất hàng** (giảm tồn kho).
  + Theo dõi và hiển thị tình trạng hàng tồn kho theo thời gian thực.
* **Kiến trúc Rõ ràng và Bảo mật:**
  + Áp dụng **Kiến trúc Lớp (Layered Architecture)** nghiêm ngặt để đảm bảo tính dễ bảo trì và khả năng mở rộng của hệ thống.
  + Triển khai cơ chế xác thực robust bằng **JWT (JSON Web Tokens)** và **BCrypt** để bảo vệ dữ liệu và phân quyền truy cập cho người dùng.
* **Giao diện Tương tác:** Cung cấp giao diện người dùng trực quan, đáp ứng bằng **JavaFX** để người vận hành có thể dễ dàng quản lý và theo dõi các hoạt động kho.

Dự án này không chỉ là một ứng dụng mô phỏng mà còn là một minh chứng thực tế cho việc áp dụng thành công các nguyên lý thiết kế và triển khai một hệ thống phân tán trong quản lý nghiệp vụ.

## 2. Cơ sở lý thuyết.

### 2.1. Kiến trúc Client-Server Phân tán

Hệ thống được thiết kế theo mô hình **Client-Server phân tán**. Trong đó:

* **Server (Máy chủ):** Đóng vai trò là trung tâm xử lý dữ liệu, chứa toàn bộ logic nghiệp vụ (Business Logic) về quản lý kho, hàng tồn kho và giao dịch nhập/xuất.
* **Client (Máy khách):** Là ứng dụng giao diện (sử dụng **JavaFX**), chịu trách nhiệm hiển thị dữ liệu và tiếp nhận tương tác từ người dùng, sau đó gửi các yêu cầu xử lý về Server.

Mô hình này cho phép tách biệt rõ ràng giữa logic nghiệp vụ (Server) và giao diện người dùng (Client), tạo điều kiện cho việc phát triển và bảo trì độc lập, đồng thời hỗ trợ khả năng mở rộng hệ thống sau này.

### 2.2. Cơ sở lý thuyết về JavaFX

**JavaFX** là một bộ công cụ phát triển ứng dụng khách đa nền tảng (Cross-Platform Rich Client Application Development Toolkit) được phát triển để thay thế cho Java Swing và AWT. Mục tiêu của JavaFX là cung cấp một nền tảng mạnh mẽ, hiện đại để xây dựng các ứng dụng có giao diện đồ họa phong phú, hiệu suất cao và nhất quán trên các hệ điều hành khác nhau (Windows, macOS, Linux).

Trong hệ thống Quản lý kho này, JavaFX được sử dụng để xây dựng **Ứng dụng Client (Máy khách)**, chịu trách nhiệm kết nối và trao đổi dữ liệu qua giao thức gRPC với Server.

#### 2.2.1. Kiến trúc Cốt lõi: Scene Graph (Đồ thị Cảnh)

Trái ngược với các thư viện GUI cũ sử dụng mô hình vẽ trực tiếp (immediate mode), JavaFX sử dụng mô hình **Scene Graph (Đồ thị Cảnh)** – một cấu trúc dữ liệu dạng cây (Tree-like Structure) giữ lại (retained mode) để tổ chức và quản lý tất cả các thành phần đồ họa của giao diện.

* **Node (Nút):** Là đơn vị cơ bản nhất trong Scene Graph. Mọi thành phần hiển thị trên màn hình (như nút nhấn, trường văn bản, bảng, hình dạng, hiệu ứng,...) đều là một **Node**.
  + **Root Node (Nút Gốc):** Nút cao nhất trong một Scene Graph.
  + **Branch Node (Nút Nhánh):** Các Node chứa các Node con khác (ví dụ: các Layout Container như VBox, GridPane).
  + **Leaf Node (Nút Lá):** Các Node không chứa Node con (ví dụ: Button, Label, TextField).
* **Ưu điểm:** Scene Graph cho phép nhà phát triển dễ dàng áp dụng các phép biến đổi (như xoay, dịch chuyển), hiệu ứng (shadow, blur), và quản lý trạng thái của các thành phần đồ họa một cách hiệu quả và tự nhiên.

#### 2.2.2. Vòng đời Ứng dụng (Application Lifecycle)

Mỗi ứng dụng JavaFX phải kế thừa lớp trừu tượng javafx.application.Application và tuân thủ một vòng đời cơ bản:

1. **init():** Phương thức tùy chọn, được gọi trước khi ứng dụng khởi động.
2. **start(Stage primaryStage):** Phương thức bắt buộc. Đây là điểm khởi đầu chính của ứng dụng.
   * **Stage (Sân khấu):** Đại diện cho cửa sổ chính của ứng dụng (Window). Stage được cung cấp bởi hệ điều hành.
   * **Scene (Cảnh):** Là một thùng chứa (Container) chứa toàn bộ Scene Graph (các Node). Một Stage phải chứa một Scene, và một Scene phải chứa một Scene Graph (bắt đầu từ một Root Node).
   * start() thiết lập Scene Graph, gắn nó vào Scene, và hiển thị Stage.
3. **stop():** Phương thức tùy chọn, được gọi khi ứng dụng kết thúc, thường dùng để dọn dẹp tài nguyên (ví dụ: đóng kết nối gRPC với Server).

#### 2.2.3. Tách biệt Giao diện và Logic (FXML, Controller, và CSS)

Để duy trì tính dễ bảo trì và mở rộng, JavaFX khuyến khích việc tách biệt hoàn toàn giữa thiết kế giao diện (View) và logic lập trình (Controller), thường được triển khai thông qua các công nghệ sau:

* **FXML**
* **Khái niệm:** Là một ngôn ngữ đánh dấu dựa trên XML, được thiết kế để khai báo và xây dựng giao diện người dùng (View) một cách rõ ràng và trực quan.
* **Vai trò:** Thay vì tạo giao diện bằng code Java thuần túy, FXML cho phép định nghĩa cấu trúc Scene Graph, các thành phần UI, và cách bố trí (Layout) của chúng.
* **Lợi ích:** Thúc đẩy mô hình **Model-View-Controller (MVC)** hoặc **Model-View-Presenter (MVP)**, giúp lập trình viên và nhà thiết kế giao diện làm việc độc lập.
* **Controller**
* Mỗi tệp FXML có thể được liên kết với một lớp Controller trong Java.
* Lớp Controller chứa **logic nghiệp vụ của giao diện**, bao gồm:
  + Xử lý sự kiện (Event Handling) từ người dùng (ví dụ: nhấn nút).
  + Tương tác với các thành phần UI (ví dụ: cập nhật dữ liệu trong TableView).
  + Trong hệ thống này, các Controller của Client sẽ chứa logic để gọi các phương thức dịch vụ gRPC (như GrpcClientService) để gửi yêu cầu và nhận dữ liệu từ Server.
* **CSS (Cascading Style Sheets)**
* JavaFX hỗ trợ áp dụng phong cách (styling) cho giao diện bằng CSS tiêu chuẩn.
* Việc này giúp tách biệt thiết kế trực quan khỏi cấu trúc và logic, cho phép thay đổi giao diện (look-and-feel) mà không cần chỉnh sửa code Java hay FXML.

#### 2.2.4. Khả năng Tương tác và Bố trí (Controls and Layouts)

* **Controls (Thành phần Điều khiển):** JavaFX cung cấp một bộ Controls phong phú (ví dụ: Button, Label, TextField, TableView, TreeView, Chart) để tạo giao diện. Đặc biệt, các Controls hiển thị dữ liệu phức tạp như TableView và Chart rất cần thiết để hiển thị trạng thái kho và báo cáo theo thời gian thực.
* **Layouts (Bộ chứa Bố cục):** Đây là các Node đặc biệt, có nhiệm vụ tự động sắp xếp và quản lý vị trí, kích thước của các Node con. Các Layout cơ bản được sử dụng trong dự án (như có thể thấy trong các file FXML) bao gồm:
  + **VBox / HBox:** Sắp xếp các thành phần theo chiều dọc/ngang.
  + **GridPane:** Sắp xếp thành phần theo lưới (hàng và cột), lý tưởng cho các biểu mẫu nhập liệu.
  + **BorderPane:** Sắp xếp thành phần vào 5 khu vực (Trên, Dưới, Trái, Phải, Giữa), thường dùng làm khung ứng dụng chính.
  + Các Layout này đảm bảo rằng giao diện người dùng **đáp ứng (responsive)** và thích ứng tốt khi cửa sổ ứng dụng được thay đổi kích thước.

### 2.3 gRPC:

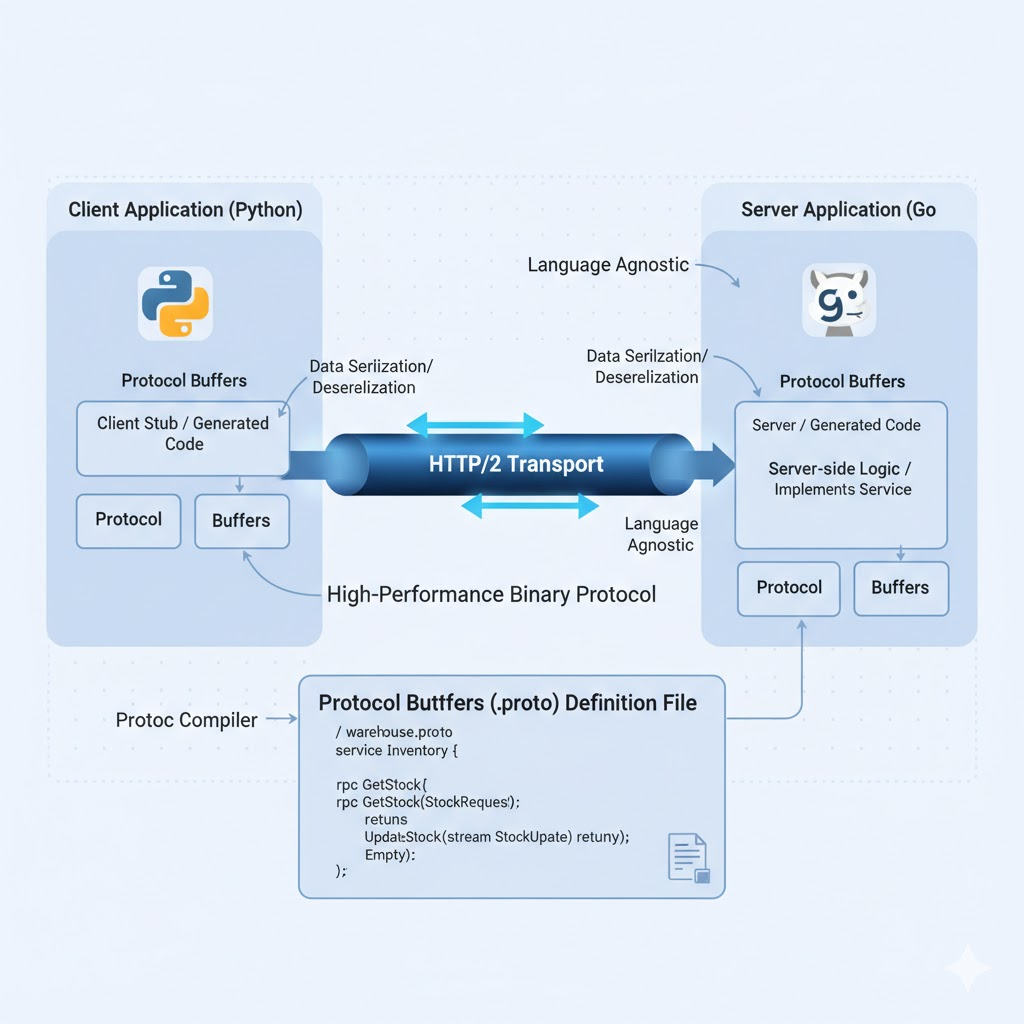
gRPC (gRPC Remote Procedure Call) là một framework RPC (Remote Procedure Call) mã nguồn mở và hiệu suất cao, được phát triển bởi Google. Nó cho phép các ứng dụng giao tiếp với nhau như thể chúng đang gọi các hàm cục bộ, bất kể chúng được triển khai ở đâu và bằng ngôn ngữ nào. gRPC được thiết kế để kết nối các dịch vụ trong môi trường microservices, đặc biệt hiệu quả trong các hệ thống phân tán.

**Đặc điểm nổi bật của gRPC:**

* **Hiệu suất cao:** gRPC sử dụng Protocol Buffers (Protobuf) làm định dạng dữ liệu và HTTP/2 làm giao thức truyền tải, mang lại hiệu suất vượt trội so với các giao thức khác như REST với JSON qua HTTP/1.1. Protobuf là một cơ chế tuần tự hóa dữ liệu hiệu quả, tạo ra các gói tin nhỏ gọn và nhanh chóng.
* **Hỗ trợ đa ngôn ngữ:** Với việc sử dụng Protobuf, gRPC cho phép bạn định nghĩa các dịch vụ và tin nhắn một lần, sau đó tạo mã client và server bằng nhiều ngôn ngữ lập trình khác nhau. Điều này rất hữu ích trong các hệ thống phân tán nơi các dịch vụ có thể được viết bằng các ngôn ngữ khác nhau.
* **Truyền tải song công (Full-duplex streaming):** gRPC hỗ trợ bốn loại hình giao tiếp:
  + **Unary RPC:** Client gửi một yêu cầu và nhận một phản hồi (như một cuộc gọi API thông thường).
  + **Server Streaming RPC:** Client gửi một yêu cầu và nhận về một luồng phản hồi từ server.
  + **Client Streaming RPC:** Client gửi một luồng yêu cầu đến server và nhận về một phản hồi duy nhất.
  + **Bidirectional Streaming RPC:** Cả client và server đều có thể gửi và nhận các luồng tin nhắn độc lập.
* Tính năng streaming này đặc biệt hữu ích cho các ứng dụng yêu cầu giao tiếp liên tục hoặc xử lý dữ liệu lớn.
* **Tích hợp tốt với hệ sinh thái Cloud Native:** gRPC là lựa chọn phổ biến cho các kiến trúc microservices và được hỗ trợ tốt bởi các công nghệ như Kubernetes.

**Cách thức hoạt động của gRPC:**

1. **Định nghĩa dịch vụ (.proto):** Bạn bắt đầu bằng việc định nghĩa các dịch vụ (service), các phương thức (method) và các cấu trúc dữ liệu (message) trong một tệp tin .proto bằng cú pháp Protocol Buffers.
2. **Tạo mã nguồn:** Trình biên dịch Protobuf (protoc) sẽ tạo ra mã nguồn client và server tự động từ tệp .proto. Mã này chứa các interface, stub (client-side) và abstract class (server-side) cần thiết để triển khai dịch vụ.
3. **Triển khai server:** Bạn triển khai logic nghiệp vụ cho các phương thức đã định nghĩa trên server.
4. **Triển khai client:** Client sử dụng các stub được tạo ra để gọi các phương thức từ xa trên server.



Hình 1 - Hình ảnh mô tả cách hoạt động của gRPC

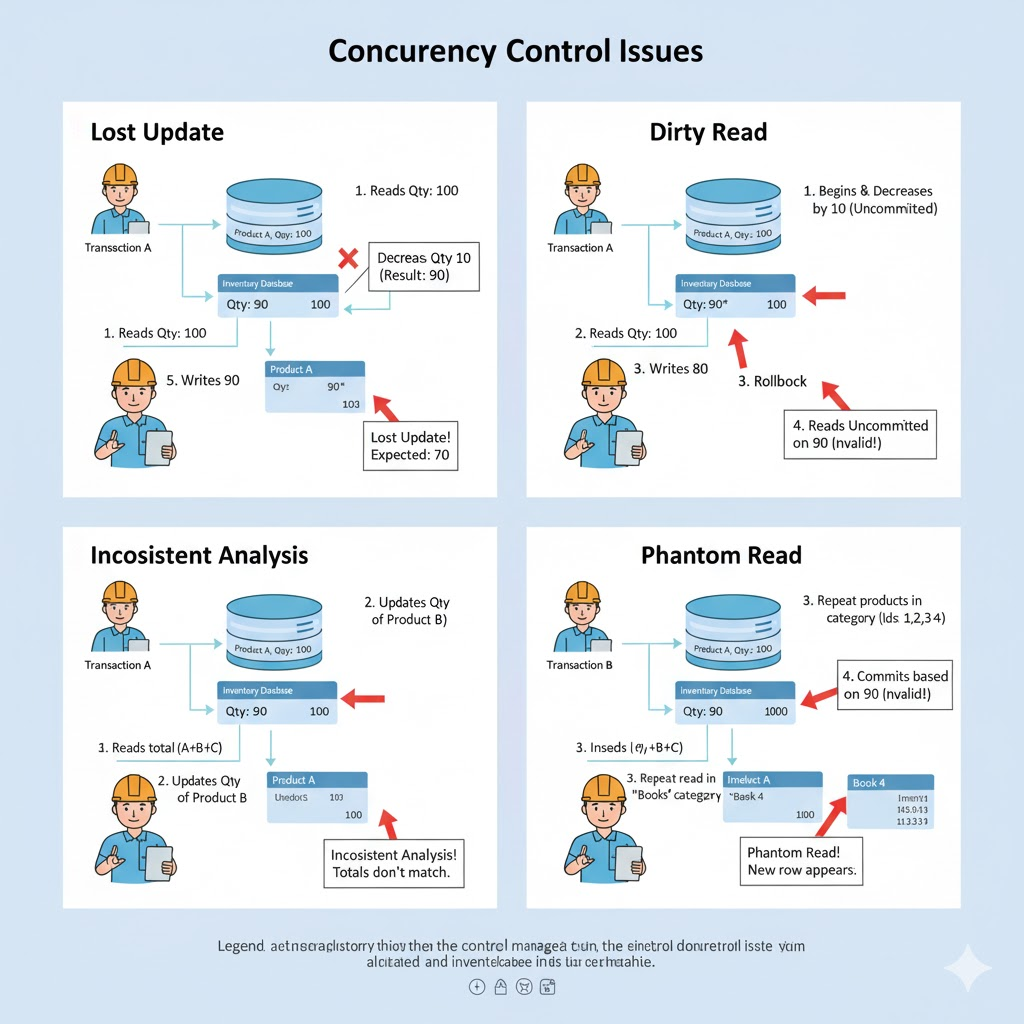
### 2.4. Xử lý Tương Tranh (Concurrency Control)

Xử lý tương tranh là một khía cạnh cực kỳ quan trọng trong các hệ thống phân tán, đặc biệt là trong quản lý kho. Nó đề cập đến các kỹ thuật và cơ chế đảm bảo tính đúng đắn và nhất quán của dữ liệu khi nhiều giao dịch (transaction) hoặc quy trình (process) cố gắng truy cập và sửa đổi cùng một tài nguyên (ví dụ: số lượng tồn kho của một sản phẩm) đồng thời.

#### 2.4.1 Các vấn đề khi không xử lý tương tranh:

Nếu không có xử lý tương tranh thích hợp, các vấn đề sau có thể xảy ra:

* **Lost Update (Mất cập nhật):** Hai giao dịch đọc cùng một dữ liệu, cả hai cùng sửa đổi, và một trong hai sửa đổi bị ghi đè bởi cái kia. Ví dụ, hai nhân viên cùng lúc cập nhật số lượng tồn kho của sản phẩm A từ 100. Người thứ nhất giảm 10, người thứ hai giảm 20. Nếu không có xử lý tương tranh, kết quả có thể là 90 hoặc 80, chứ không phải 70 như mong đợi.
* **Dirty Read (Đọc bẩn):** Một giao dịch đọc dữ liệu chưa được cam kết (uncommitted data) của một giao dịch khác. Nếu giao dịch ghi sau đó bị rollback, dữ liệu mà giao dịch đọc đã đọc sẽ không còn hợp lệ.
* **Inconsistent Analysis (Phân tích không nhất quán):** Một giao dịch đọc một tập hợp dữ liệu, trong khi đó một giao dịch khác thay đổi một phần của tập hợp đó, dẫn đến việc giao dịch đọc nhận được dữ liệu không nhất quán.
* **Phantom Read (Đọc "ma"):** Một giao dịch thực hiện một truy vấn hai lần, và lần thứ hai trả về một tập hợp các hàng khác nhau do một giao dịch khác đã chèn hoặc xóa hàng giữa hai lần đọc.



Hình 2 - Các tác hại nếu không xử lý tương tranh

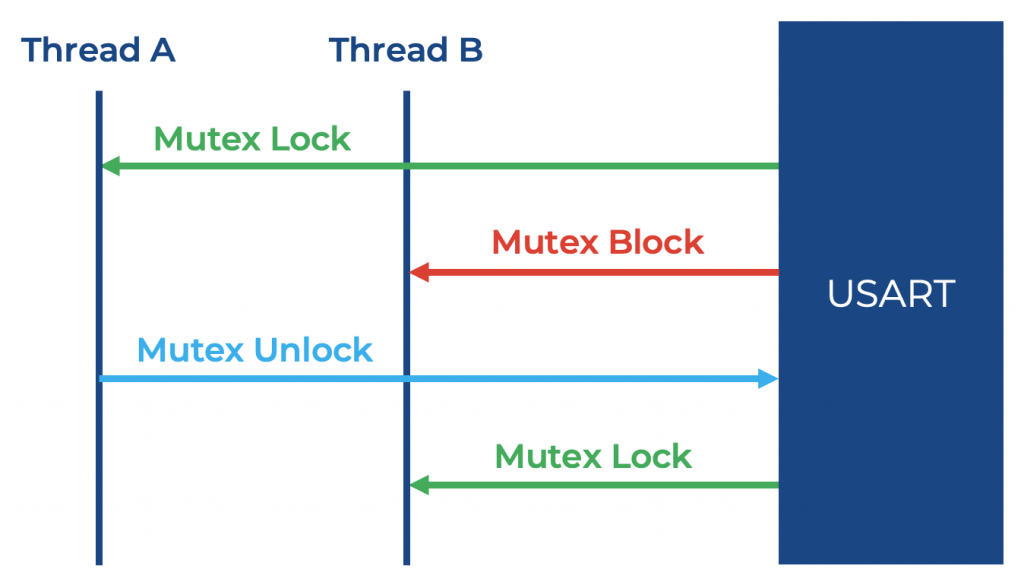
#### 2.4.1 Cơ chế đồng bộ hóa trong Lập trình đa luồng:

Trong các hệ thống phân tán mà Server phải xử lý hàng trăm yêu cầu đồng thời, việc quản lý truy cập vào tài nguyên chia sẻ (shared resources) trong bộ nhớ là vô cùng quan trọng. Có hai cơ chế đồng bộ hóa phổ biến nhất là **Mutex** và **Read-Write Lock**.

##### 1. Mutex (Mutual Exclusion)

Mutex (Khóa loại trừ lẫn nhau) là cơ chế đơn giản nhất để ngăn chặn điều kiện đua (race condition),

* **Nguyên lý:** Nó hoạt động như một chiếc chìa khóa duy nhất. Tại một thời điểm, chỉ có **một luồng (thread)** được phép giữ khóa và truy cập tài nguyên. Các luồng khác muốn truy cập phải xếp hàng chờ.



Hình 3 - Sơ đồ tuần tự thể hiện cơ chế Mutex

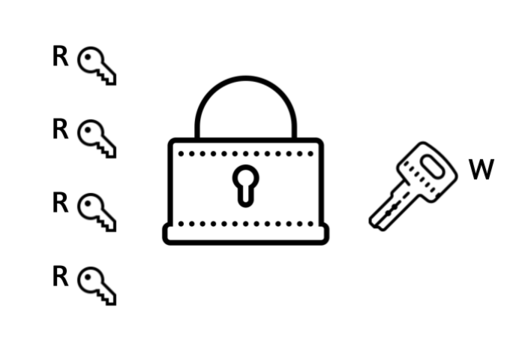


Hình 4 - Hình ảnh minh họa mutex

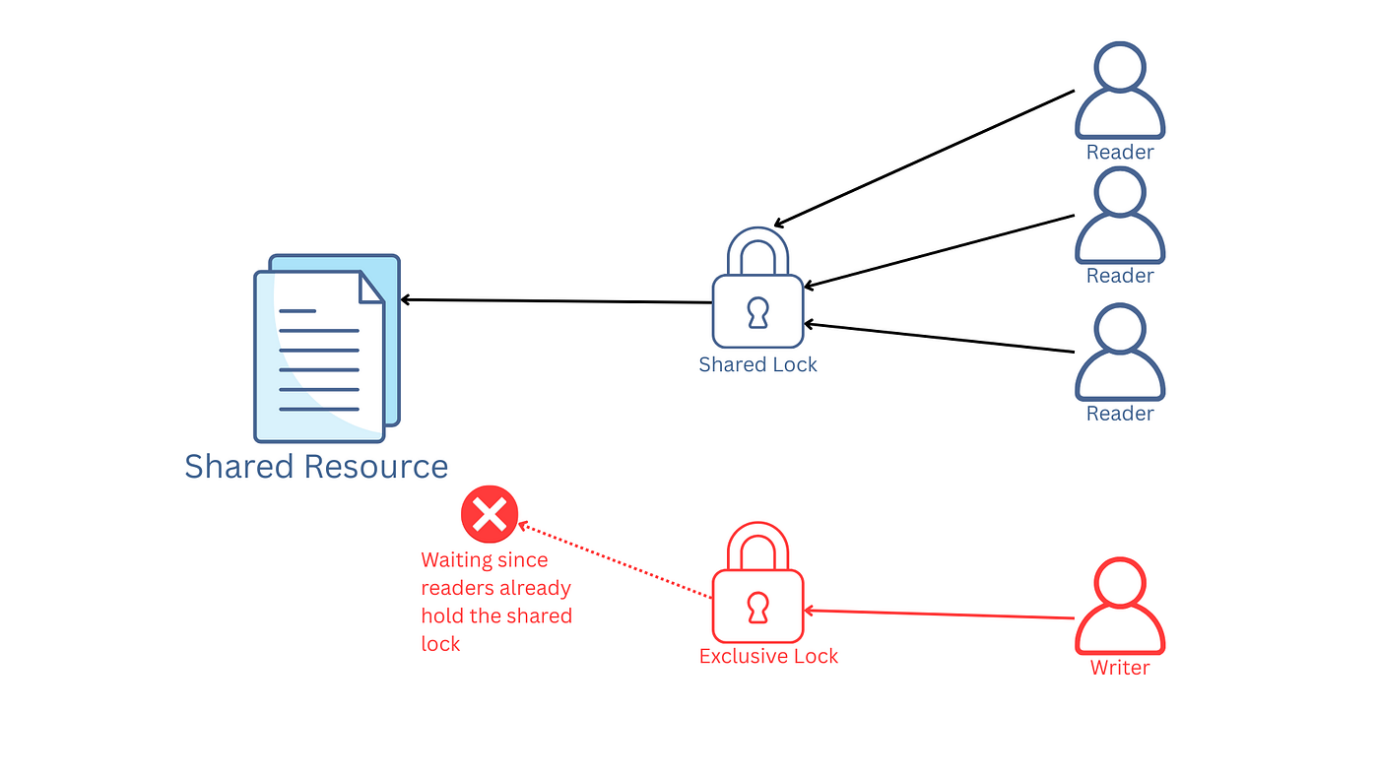
* **Nhược điểm:** Mutex đối xử công bằng với mọi thao tác Đọc và Ghi. Ngay cả khi nhiều luồng chỉ muốn *đọc* dữ liệu (hành động an toàn), chúng cũng bị chặn lại, dẫn đến hiệu suất thấp trong các hệ thống có tần suất đọc cao.

##### 2. Read-Write Lock (Khóa Đọc-Ghi)

Read-Write Lock là một cơ chế khóa nâng cao, được thiết kế để giải quyết bài toán **"Readers-Writers Problem"**. Nó phân biệt rõ ràng giữa hai loại quyền truy cập:



Hình 5 - Hình ảnh minh họa Read-Write Lock



Hình 6 - Hình ảnh mô tả luồng hoạt động của Read-Write Lock

1. **Khóa Đọc (Read Lock / Shared Lock):**
   * Cho phép **nhiều luồng** cùng giữ khóa đọc đồng thời.
   * Điều kiện: Không có luồng nào đang giữ Khóa Ghi.
   * *Ý nghĩa:* Tăng tối đa thông lượng (throughput) cho các tác vụ tra cứu, xem báo cáo.
2. **Khóa Ghi (Write Lock / Exclusive Lock):**
   * Chỉ cho phép **duy nhất một luồng** giữ khóa ghi tại một thời điểm.
   * Khi một luồng giữ Khóa Ghi, không luồng nào khác (kể cả đọc hay ghi) được phép truy cập.
   * *Ý nghĩa:* Đảm bảo tính toàn vẹn dữ liệu tuyệt đối khi có thay đổi.

**Bảng so sánh hiệu năng:**

|  |  |  |
| --- | --- | --- |
| **Đặc điểm** | **Mutex (Khóa đơn)** | **Read-Write Lock** |
| **Cho phép Đọc song song** | Không (Tuần tự) | **Có** (Song song) |
| **Cho phép Ghi song song** | Không | Không |
| **Tính phù hợp** | Hệ thống Ghi nhiều | **Hệ thống Đọc nhiều, Ghi ít** |
| **Độ phức tạp** | Thấp | Trung bình |

Trong ngữ cảnh của hệ thống quản lý kho, nơi thao tác xem danh sách sản phẩm (Đọc) diễn ra thường xuyên hơn rất nhiều so với thao tác nhập/xuất kho (Ghi), **Read-Write Lock** là sự lựa chọn tối ưu về mặt lý thuyết lẫn thực tiễn.

### 2.5. Inversion of Control (IoC) và Dependency Injection (DI)

Trong phát triển phần mềm hiện đại, việc quản lý sự phụ thuộc giữa các thành phần (components) là yếu tố then chốt để tạo ra một hệ thống linh hoạt, dễ bảo trì và dễ kiểm thử. Để giải quyết vấn đề này, hai khái niệm **Inversion of Control (IoC)** và **Dependency Injection (DI)** đã ra đời và trở thành nền tảng của nhiều kiến trúc phần mềm.

#### 2.5.1. Khái niệm Inversion of Control (IoC):

**Inversion of Control (Đảo ngược điều khiển)** là một nguyên lý thiết kế, trong đó luồng điều khiển của chương trình được đảo ngược so với lập trình truyền thống.

* **Lập trình truyền thống:** Mã nguồn của chúng ta "gọi" các thư viện và tự quyết định khi nào khởi tạo, khi nào hủy các đối tượng phụ thuộc.
* **Với IoC:** Quyền điều khiển này được chuyển giao cho một thành phần bên ngoài (thường gọi là Container hoặc Framework). Thay vì đối tượng A tự tạo ra đối tượng B để sử dụng, đối tượng A sẽ "nhận" đối tượng B từ Container.

Một phép ẩn dụ nổi tiếng cho IoC là **"Hollywood Principle"**: *"Don't call us, we'll call you"* (Đừng gọi cho chúng tôi, chúng tôi sẽ gọi cho bạn).

#### 2.5.2. Dependency Injection (DI):

**Dependency Injection (Tiêm phụ thuộc)** là một mẫu thiết kế (Design Pattern) cụ thể để hiện thực hóa nguyên lý IoC. Nó đề cập đến việc cung cấp (tiêm) các đối tượng phụ thuộc (dependencies) vào một đối tượng khác thay vì để đối tượng đó tự tạo ra chúng.

Có 3 cách phổ biến để thực hiện DI:

1. **Constructor Injection (Tiêm qua hàm khởi tạo):** Các phụ thuộc được truyền vào qua constructor của lớp. Đây là cách an toàn và được khuyến khích nhất vì nó đảm bảo đối tượng luôn ở trạng thái hợp lệ sau khi khởi tạo.
2. **Setter Injection (Tiêm qua hàm Setter):** Các phụ thuộc được truyền vào qua các hàm .
3. **Interface Injection:** Ít phổ biến hơn, sử dụng một interface để cung cấp phương thức nhận phụ thuộc.

**Ví dụ minh họa:**

**Cách truyền thống (Coupled - Gắn kết chặt):**Java  
public class ProductService {

// ProductService phụ thuộc trực tiếp vào ProductRepository cụ thể

private ProductRepository repo = new ProductRepository();

public void addProduct(Product p) {

repo.save(p);

}

}

* *Vấn đề:* Khó thay đổi repo (ví dụ muốn đổi sang lưu file thay vì DB), khó viết Unit Test vì không thể thay thế repo bằng mock object.

**Cách dùng DI (Decoupled - Gắn kết lỏng):**Java  
public class ProductService {

private ProductRepository repo;

// Phụ thuộc được "tiêm" vào từ bên ngoài

public ProductService(ProductRepository repo) {

this.repo = repo;

}

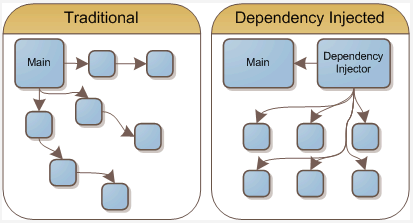
public void addProduct(Product p) {

repo.save(p);

}

}

* *Lợi ích:* ProductService không cần biết repo được tạo ra như thế nào. Chúng ta có thể dễ dàng truyền vào FileProductRepository hay MockProductRepository tùy ngữ cảnh.



Hình 7 - Minh họa phương pháp truyền thống và IoC/DI

#### 2.5.3. Lợi ích của IoC và DI trong Hệ thống Phân tán

1. **Giảm sự phụ thuộc (Decoupling):** Các lớp (Service, Repository) hoạt động độc lập với việc khởi tạo các lớp khác. Điều này giúp dễ dàng thay đổi một thành phần mà không ảnh hưởng đến toàn bộ hệ thống.
2. **Dễ dàng kiểm thử (Testability):** Khi viết Unit Test cho ProductService, ta có thể dễ dàng tạo một "Repository giả" (Mock) và truyền vào constructor để kiểm tra logic nghiệp vụ mà không cần thao tác với file hay CSDL thật.
3. **Quản lý vòng đời đối tượng (Lifecycle Management):** Trong các hệ thống lớn, IoC Container giúp quản lý việc tạo, sử dụng lại (Singleton) và hủy các đối tượng một cách tập trung và hiệu quả.

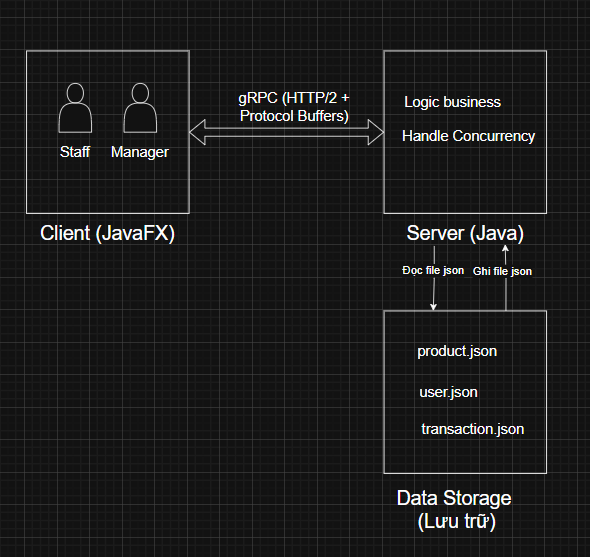
# CHƯƠNG II. THIẾT KẾ HỆ THỐNG

Chương này trình bày các quyết định kiến trúc ở mức cao, phác thảo "bản thiết kế" của hệ thống, trả lời cho câu hỏi **"Cái gì"** (What) và **"Tại sao"** (Why) hệ thống được xây dựng như vậy.

## 1. Kiến trúc Tổng quan :

Hệ thống được thiết kế theo mô hình **Client-Server** phân tán, được tối ưu hóa cho nghiệp vụ quản lý kho. Kiến trúc này bao gồm các thành phần cốt lõi:

* **Client (JavaFX):** Một ứng dụng "dày" (thick client) được xây dựng bằng JavaFX, chịu trách nhiệm cho toàn bộ giao diện người dùng (GUI) và trải nghiệm tương tác. Client giao tiếp với Server thông qua các lời gọi thủ tục từ xa.
* **Server (Java):** Một ứng dụng Java độc lập, đóng vai trò là "não bộ" của hệ thống. Server xử lý toàn bộ logic nghiệp vụ, xác thực, phân quyền và là thành phần duy nhất được phép truy cập vào kho dữ liệu.
* **Kênh Giao tiếp (gRPC):** Việc giao tiếp giữa Client và Server được thực hiện độc quyền qua gRPC. Lựa chọn này dựa trên hiệu suất vượt trội của gRPC (sử dụng HTTP/2 và Protocol Buffers) và khả năng định nghĩa dịch vụ (service contract) một cách chặt chẽ thông qua các tệp .proto.
* **Lưu trữ Dữ liệu (JSON Files):** Thay vì sử dụng CSDL quan hệ hoặc NoSQL, hệ thống sử dụng một tập hợp các tệp tin JSON phẳng (users.json, products.json, history.json) để lưu trữ toàn bộ trạng thái.

****

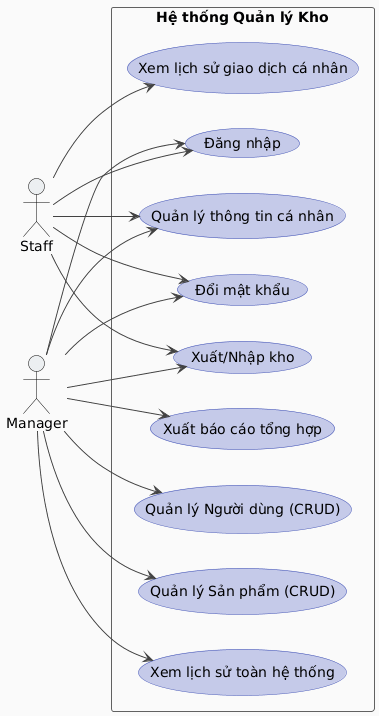
Hình 8 - Sơ đồ các thành phần của hệ thống

## 2. Phân tích Tác nhân và Phân quyền :

Để mô hình hóa các yêu cầu nghiệp vụ, hệ thống định nghĩa hai tác nhân (actor) chính, tương ứng với hai vai trò (role) có quyền hạn khác nhau. Thiết kế này là nền tảng cho việc triển khai **Kiểm soát Truy cập Dựa trên Vai trò (Role-Based Access Control - RBAC)**:

* **Manager (Quản lý):**
  + **Quản lý Hệ thống:** Có toàn quyền CRUD (Tạo, Đọc, Cập nhật, Xóa) đối với Sản phẩm; Quản lý tài khoản người dùng (thêm Staff, kích hoạt/vô hiệu hóa tài khoản).
  + **Nghiệp vụ Kho:** Có thể thực hiện các nghiệp vụ cơ bản như xuất/nhập kho (giống Staff).
  + **Giám sát & Báo cáo:** Có thể xem lịch sử giao dịch của *toàn bộ* hệ thống và xuất các báo cáo tổng hợp.
  + **Cá nhân:** Quản lý thông tin cá nhân và đổi mật khẩu.
* **Staff (Nhân viên):**
  + **Nghiệp vụ Kho:** Chỉ có quyền thực hiện các nghiệp vụ chính là xuất kho và nhập kho.
  + **Giám sát:** Chỉ được phép xem lịch sử giao dịch của *chính bản thân* mình.
  + **Cá nhân:** Quản lý thông tin cá nhân và đổi mật khẩu.

Việc phân quyền rõ ràng này sẽ được thực thi ở phía Server bằng một Interceptor để đảm bảo tính bảo mật và đúng đắn của nghiệp vụ.



Hình 9 - Sơ đồ usecase tổng quát các chức năng của 2 tác nhân Staff và Manager.

## 3. Thiết kế Giao tiếp Dịch vụ (gRPC Service Contract) :

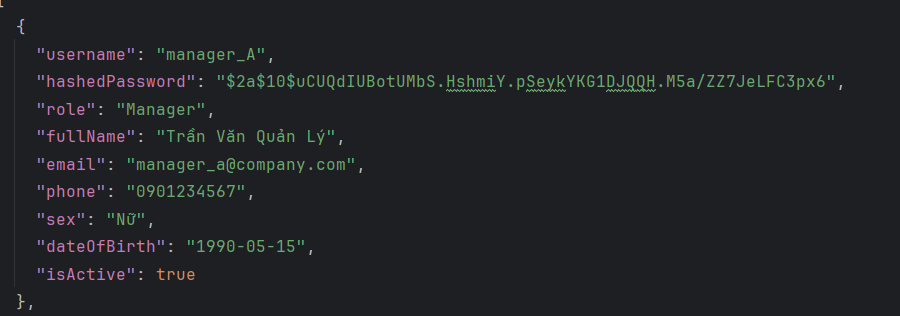
Dựa trên phân tích vai trò, "hợp đồng" (contract) giao tiếp được định nghĩa trong các tệp .proto, chia thành các dịch vụ (services) logic, tuân thủ Nguyên tắc Phân tách Trách nhiệm (Separation of Concerns):

* **AuthService (auth.proto):** Xử lý các nghiệp vụ chung cho cả hai vai trò (Đăng nhập, xem thông tin cá nhân, đổi mật khẩu).
* **UserManagementService (user.proto):** Dịch vụ dành riêng cho Manager để quản lý tài khoản người dùng.
* **ProductManagementService (product.proto):** Dịch vụ dành riêng cho Manager để thực hiện CRUD sản phẩm.
* **WarehouseService (warehouse.proto):** Xử lý các nghiệp vụ cốt lõi về kho (xuất, nhập, xem lịch sử, xem dashboard) cho cả Staff và Manager.

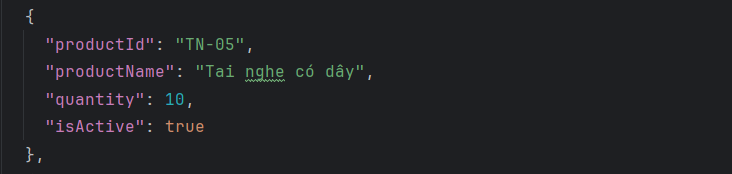
## 4. Thiết kế Lưu trữ Dữ liệu (Data Storage Design) :

Đây là một trong những quyết định thiết kế then chốt của dự án.

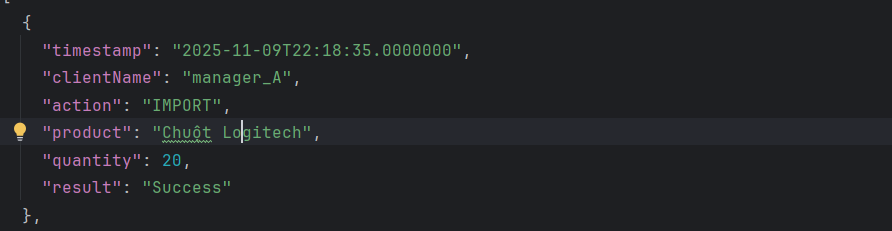
* **Lựa chọn:** Hệ thống sử dụng các tệp tin **JSON** làm cơ sở dữ liệu. Toàn bộ trạng thái của ứng dụng (thông tin người dùng, danh sách sản phẩm và số lượng, lịch sử giao dịch) được tuần tự hóa (serialized) và lưu trữ trực tiếp trên hệ thống tệp tin của server.
* **Ưu điểm:**
  + **Đơn giản hóa (Simplicity):** Loại bỏ hoàn toàn sự phức tạp của việc cài đặt, cấu hình và bảo trì một Hệ quản trị CSDL bên ngoài (ví dụ: MySQL, PostgreSQL).
  + **Tính di động (Portability):** Hệ thống có thể chạy ngay lập tức trên bất kỳ máy nào có JRE mà không cần cài đặt thêm. Toàn bộ "CSDL" chỉ là một thư mục data/.
  + **Dễ gỡ lỗi (Debuggability):** Dữ liệu ở định dạng con người có thể đọc (human-readable), cho phép kiểm tra và chỉnh sửa thủ công khi cần thiết.
* **Thách thức Thiết kế:** Quyết định này đánh đổi sự đơn giản lấy một thách thức lớn: **Điều khiển tương tranh (Concurrency Control)**.
* Các đối tượng trong Data Storage:



Hình 10 - Cấu trúc dữ liệu cho entity User



Hình 11 - Cấu trúc dữ liệu cho entity Product



Hình 12 - Cấu trúc dữ liệu cho entity Transaction

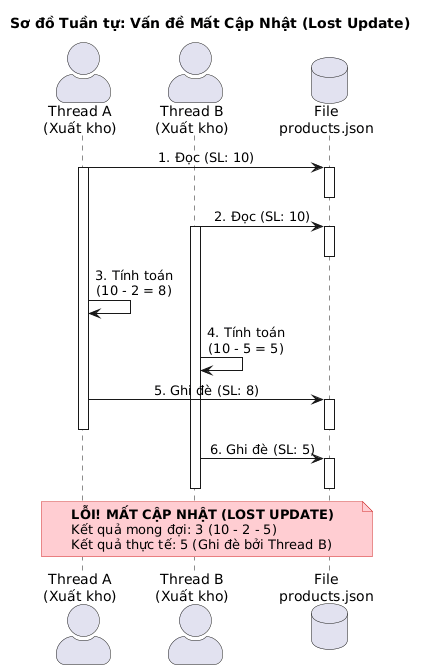
## 5. Thách thức Cốt lõi: Rủi ro Tương tranh với File JSON

Không giống như các CSDL chuyên dụng (cung cấp các cơ chế khóa ở mức bản ghi - *row-level locking* hoặc MVCC), hệ thống tệp tin không có cơ chế nào để ngăn chặn hai tiến trình (process) hoặc luồng (thread) cùng thao tác trên một tệp.

Khi Server gRPC (vốn được thiết kế để xử lý đa luồng) nhận hai yêu cầu *ghi* (ví dụ: exportStock) cùng một lúc, nó sẽ tạo ra hai luồng xử lý song song. Cả hai luồng này sẽ thực hiện chu trình "Đọc-Sửa-Ghi":

1. **Luồng A** đọc products.json (ví dụ: Sản phẩm X, SL: 10).
2. **Luồng B** cũng đọc products.json (ví dụ: Sản phẩm X, SL: 10).
3. **Luồng A** tính toán trong bộ nhớ (10 - 2 = 8) và **ghi đè** toàn bộ tệp products.json với SL=8.
4. **Luồng B** tính toán trong bộ nhớ (10 - 5 = 5) và **ghi đè** toàn bộ tệp products.json với SL=5.

Kết quả là cập nhật của Luồng A bị **Mất (Lost Update)**. Dữ liệu kho bị sai lệch nghiêm trọng (SL thực tế là 5, trong khi đáng lẽ phải là 3).

****

Hình 13 - Sơ đồ tuần tự thể hiện việc mất dữ liệu khi không xử lý tương tranh

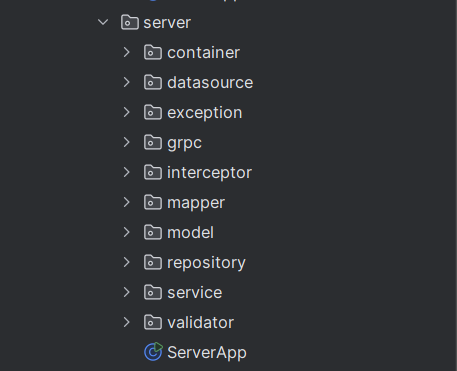
Do đó, một giải pháp khóa ở cấp độ ứng dụng (application-level lock) là **bắt buộc** phải được thiết kế để đảm bảo mọi thao tác ghi vào tệp JSON phải diễn ra một cách tuần tự và nguyên tử (atomic).

# CHƯƠNG III. TRIỂN KHAI HỆ THỐNG

Chương này mô tả **"Cách thức"** (How) bản thiết kế ở Chương 2 đã được hiện thực hóa bằng mã nguồn Java, tập trung vào các giải pháp kỹ thuật cụ thể để giải quyết các thách thức đã đặt ra.

## 1. Kiến trúc Lớp (Layered Architecture) phía Server

Để quản lý sự phức tạp của ứng dụng mà không cần đến framework (như Spring), Server được triển khai theo kiến trúc N-Lớp (N-tier) và Nguyên tắc Phân tách Trách nhiệm. Cấu trúc 10 thư mục thể hiện rõ ràng kiến trúc này:

****

Hình 14 - Cấu trúc thư mục Server

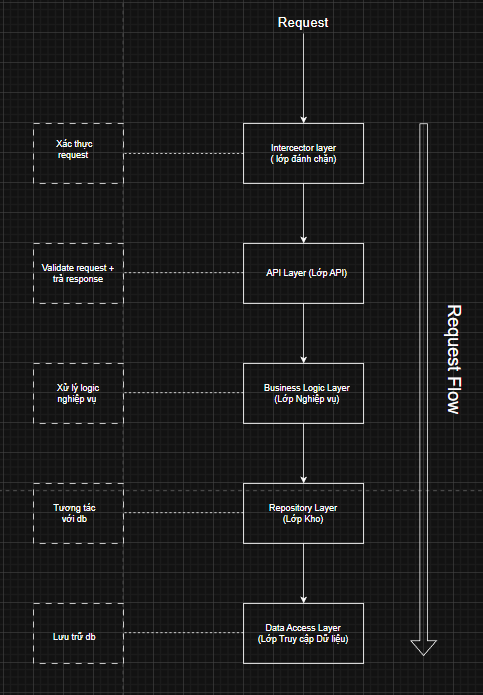
* **container:** Để áp dụng nguyên lý **IoC và DI** đã trình bày ở cơ sở lý thuyết mà không cần phụ thuộc vào các Framework nặng nề như Spring, nhóm đã tự triển khai một IoC Container đơn giản thông qua lớp **ApplicationContainer**.

Lớp này đóng vai trò là nơi duy nhất chịu trách nhiệm khởi tạo và "lắp ráp" các thành phần của hệ thống:

1. Khởi tạo DataSource (Singleton).
2. Khởi tạo Repository và tiêm DataSource vào.
3. Khởi tạo Service và tiêm Repository vào.

Việc sử dụng **Constructor Injection** giúp mã nguồn của các lớp Service và Repository hoàn toàn tách biệt, trong sáng và dễ bảo trì.

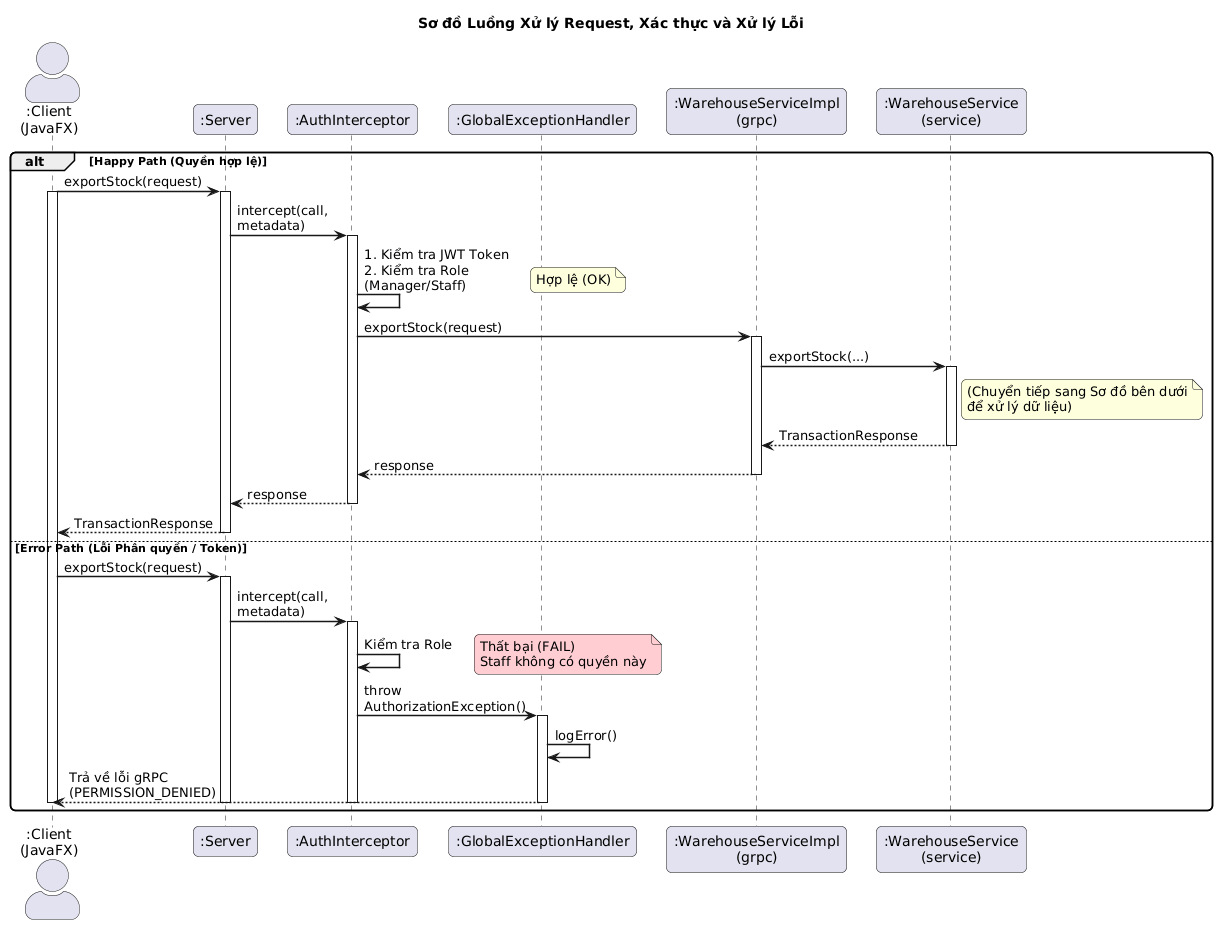
* **grpc:** Lớp "Cổng vào" (Entry Point), tương tự như Controller trong MVC. Các lớp ...ServiceImpl (ví dụ: WarehouseServiceImpl.java) kế thừa từ mã gRPC-base, nhận các đối tượng request, gọi validator và sau đó ủy quyền cho service.
* **service:** Lớp logic nghiệp vụ cốt lõi (Core Business Logic). Đây là nơi xử lý các quy tắc (ví dụ: exportStock phải kiểm tra số lượng tồn, tạo bản ghi Transaction). Lớp này chỉ nói chuyện với repository.
* **repository:** Lớp trừu tượng hóa dữ liệu (Data Abstraction Layer). Nó định nghĩa các phương thức nghiệp vụ (ví dụ: findProductById, saveUser) mà service cần. Nó làm việc với các đối tượng model của server.
* **datasource:** Lớp truy cập dữ liệu (Data Access Layer). Đây là lớp *duy nhất* được phép đọc/ghi file JSON (sử dụng thư viện Jackson/Gson). Đây cũng là nơi triển khai **giải pháp khóa tương tranh**.
* **interceptor:** Nơi xử lý các mối quan tâm "cắt ngang" (Cross-Cutting Concerns).
  + AuthInterceptor.java: Đánh chặn mọi yêu cầu, đọc JWT token từ metadata, xác thực (Authentication) và kiểm tra vai trò (Authorization) trước khi cho phép request đi tiếp.
  + GlobalExceptionHandlerInterceptor.java: Bắt (catch) tất cả các exception nghiệp vụ (ví dụ: ValidationException, ResourceNotFoundException) và chuyển đổi chúng thành các lỗi gRPC chuẩn (ví dụ: INVALID\_ARGUMENT, NOT\_FOUND), giúp client xử lý lỗi nhất quán.
* **mapper:** Các lớp tiện ích để chuyển đổi (map) dữ liệu giữa 3 lớp: đối tượng gRPC (.proto) ⇔ đối tượng Model server (.java) ⇔ đối tượng Model client (.java, nếu cần).
* **validator:** Các lớp đơn giản để kiểm tra tính hợp lệ của đầu vào (ví dụ: AddUserRequest phải có username và password không được rỗng).
* **exception:** Định nghĩa các lớp ngoại lệ nghiệp vụ tùy chỉnh (ví dụ: AuthorizationException) để service có thể "ném" (throw) và GlobalExceptionHandlerInterceptor có thể "bắt" (catch).
* **model** : chưa các class entity đại diện cho các dữ liệu được lưu trữ gồm có product, transaction và user.



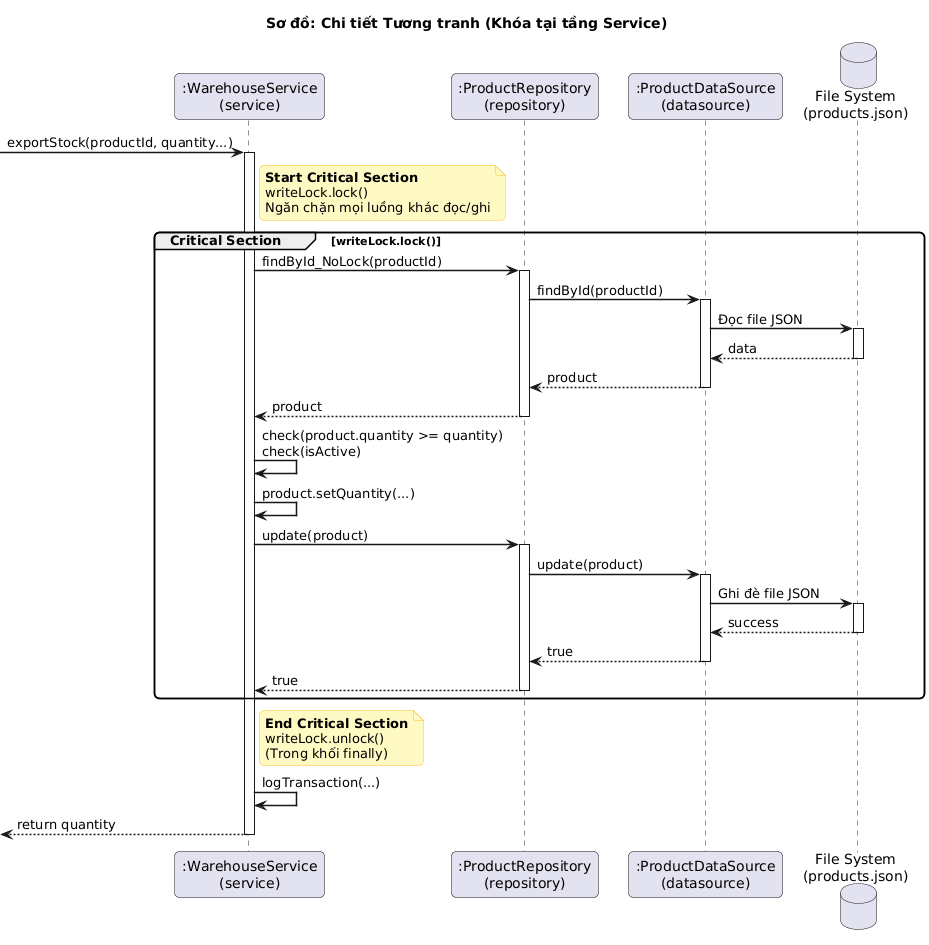
Hình 15 - Kiến trúc tổng quát server.

## 2. Luồng Xử lý Request Chi tiết (Request Flow with Interceptors) :

Sự kết hợp của các lớp trên tạo ra một luồng xử lý request (request lifecycle) mạnh mẽ và có kiểm soát.



Hình 16 - Luồng xử lý Request, Xác thực và Xử lý lỗi



Hình 17 - Luồng hoạt động về xuất sản phẩm và có mô tả về cách xử lý tương tranh.

Một luồng xử lý "thành công" (happy path) cho nghiệp vụ exportStock sẽ diễn ra như sau:

1. **Khởi tạo Yêu cầu từ Client:** Người dùng (Staff hoặc Manager) nhấn nút "Xuất kho". Client (JavaFX) sẽ tạo một yêu cầu TransactionRequest. Lúc này, SessionManager tự động đính kèm **JWT Token** vào header của yêu cầu để xác thực danh tính người gọi.
2. **Xác thực và Phân quyền (Authentication & Authorization):** Yêu cầu đến Server và bị chặn lại bởi **AuthInterceptor**. Tại đây, hệ thống thực hiện hai kiểm tra quan trọng:
   * **Xác thực:** Kiểm tra tính hợp lệ của JWT Token.
   * **Phân quyền:** Xác minh xem vai trò của người dùng (Staff/Manager) có quyền thực hiện hành động EXPORT hay không. Nếu hợp lệ, yêu cầu được chuyển tiếp; nếu không, một ngoại lệ AuthorizationException sẽ được ném ra.
3. **Xử lý Lỗi Tập trung:** Yêu cầu đi qua lớp bảo vệ thứ hai là **GlobalExceptionHandlerInterceptor**. Interceptor này bao bọc toàn bộ quá trình xử lý trong một khối try-catch toàn cục. Nhiệm vụ của nó là bắt mọi ngoại lệ nghiệp vụ (như thiếu hàng, lỗi kết nối) xảy ra ở các tầng dưới và chuyển đổi chúng thành mã lỗi gRPC chuẩn (ví dụ: INVALID\_ARGUMENT, NOT\_FOUND) để trả về cho Client một cách thân thiện.
4. **Kiểm tra Dữ liệu Đầu vào (Validation):** Tại tầng WarehouseServiceImpl, **WarehouseRequestValidator** được gọi để kiểm tra tính hợp lệ của dữ liệu gửi lên (ví dụ: số lượng xuất phải lớn hơn 0, mã sản phẩm không được rỗng).
5. **Xử lý Logic Nghiệp vụ và Khóa An toàn (Service Layer & Locking):** Đây là trái tim của quy trình. Tại phương thức exportStock của WarehouseService, một cơ chế **Khóa Bi quan (Pessimistic Locking)** được kích hoạt ngay lập tức bằng writeLock.lock(). Toàn bộ chuỗi hành động sau đây được thực hiện trong một **Vùng Găng (Critical Section)** độc quyền:
   * **Đọc dữ liệu:** Gọi ProductRepository để lấy thông tin sản phẩm mới nhất từ file JSON. Vì đang giữ khóa độc quyền, dữ liệu đọc được đảm bảo là chính xác nhất.
   * **Kiểm tra Logic:** Kiểm tra các điều kiện nghiệp vụ như: sản phẩm có đang hoạt động (isActive)? Số lượng tồn kho có đủ để xuất không?
   * **Cập nhật:** Nếu thỏa mãn, trừ số lượng tồn kho trong bộ nhớ.
   * **Ghi dữ liệu:** Gọi ProductRepository để lưu trạng thái mới xuống ProductDataSource. DataSource sẽ ghi đè toàn bộ file products.json để cập nhật thay đổi.

Chỉ khi toàn bộ quá trình trên hoàn tất (hoặc gặp lỗi), khóa mới được giải phóng (writeLock.unlock()) trong khối finally, cho phép các luồng khác tiếp tục truy cập.

1. **Ghi Nhật ký và Phản hồi:** Sau khi mở khóa, hệ thống ghi lại giao dịch vào lịch sử (Transaction Log) và trả về đối tượng TransactionResponse cho Client. Client nhận kết quả và cập nhật giao diện hiển thị số lượng tồn kho mới.

## 3. Giải pháp Tương tranh và Quản lý Khóa

### 3.1. Phân tích các chiến lược đồng bộ hóa

Trong quá trình thiết kế cơ chế truy cập đồng thời vào kho dữ liệu (Data Store), chúng tôi đã cân nhắc ba chiến lược khóa phổ biến dựa trên đặc thù của hệ thống lưu trữ hiện tại (lưu trữ trên tệp JSON và nạp toàn bộ vào HashMap trong bộ nhớ):

**1. Khóa chi tiết (Fine-grained Locking - Khóa trên từng ID sản phẩm)**

* **Cơ chế:** Cấp phát một đối tượng khóa riêng biệt cho mỗi sản phẩm. Các luồng chỉ bị chặn khi thao tác trên cùng một sản phẩm.
* **Đánh giá:** Mặc dù phương pháp này giảm thiểu sự cạnh tranh tài nguyên (contention) và tối ưu hóa việc ghi song song trên các bản ghi khác nhau, nhưng nó **không khả thi** đối với kiến trúc hiện tại vì hai lý do:
  + **Tính toàn vẹn của cấu trúc Map:** Dữ liệu được lưu trong java.util.HashMap (không luồng an toàn). Các thao tác thêm/xóa phần tử có thể thay đổi cấu trúc nội tại của Map (như *rehashing*). Việc chỉ khóa một phần tử đơn lẻ không ngăn được lỗi ConcurrentModificationException hoặc hư hỏng dữ liệu khi cấu trúc Map bị thay đổi bởi luồng khác.
  + **Cơ chế lưu trữ tệp (File Persistence):** Hệ thống sử dụng cơ chế ghi đè toàn bộ tệp products.json để lưu dữ liệu. Đây là một thao tác nguyên tử (atomic) ở cấp độ tệp tin. Dù chỉ sửa đổi một sản phẩm, hệ thống vẫn cần trạng thái nhất quán của **toàn bộ** danh sách để thực hiện quy trình tuần tự hóa (serialization) và ghi xuống đĩa.

**2. Phân đoạn khóa (Lock Striping)**

* **Cơ chế:** Chia Map thành nhiều đoạn (segments) và khóa theo từng đoạn (tương tự cơ chế của ConcurrentHashMap).
* **Đánh giá:** Phương pháp này giảm bớt độ lớn của khóa so với khóa toàn cục. Tuy nhiên, do hạn chế về thao tác I/O (ghi tệp) đã nêu trên, việc tập hợp dữ liệu từ các phân đoạn để ghi xuống tệp vẫn đòi hỏi một cơ chế đồng bộ hóa phức tạp hơn để đảm bảo tính nhất quán (snapshot) tại thời điểm ghi. Do đó, độ phức tạp triển khai không tương xứng với hiệu năng thu được trong quy mô bài toán này.

**3. Khóa độc quyền toàn cục (Global Mutex - ReentrantLock)**

* **Cơ chế:** Sử dụng một khóa duy nhất cho toàn bộ kho dữ liệu. Mọi thao tác Đọc hoặc Ghi đều phải xếp hàng tuần tự.
* **Đánh giá:** Đảm bảo an toàn tuyệt đối cho cấu trúc HashMap và thao tác ghi tệp. Tuy nhiên, phương pháp này gây lãng phí tài nguyên nghiêm trọng trong các nghiệp vụ Đọc (chiếm đa số), biến hệ thống thành xử lý đơn luồng.

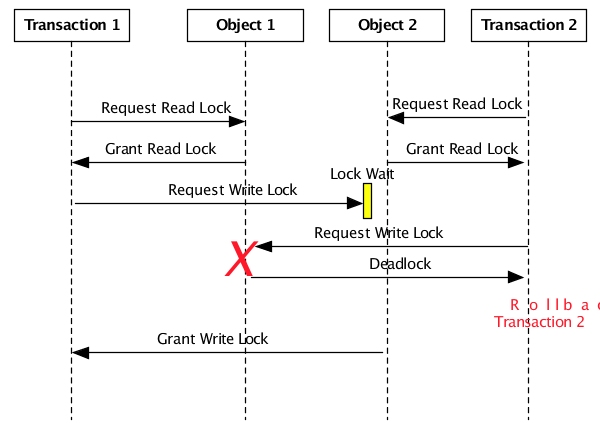
### 3.2. Giải pháp đề xuất: Global ReadWriteLock

Dựa trên phân tích trên, giải pháp tối ưu để cân bằng giữa **tính an toàn của cấu trúc dữ liệu** và **hiệu năng hệ thống** là sử dụng java.util.concurrent.locks.ReadWriteLock áp dụng ở phạm vi toàn cục (Coarse-grained).

**Lý do lựa chọn:**

1. **Phù hợp với cấu trúc HashMap:** Do HashMap yêu cầu sự bảo vệ toàn vẹn khi có thay đổi cấu trúc, việc sử dụng một khóa to bao trùm (Global Lock) là bắt buộc để đảm bảo an toàn bộ nhớ (Memory Consistency).
2. **Tối ưu hóa cho mô hình "Read-Heavy":** Hệ thống thương mại điện tử có đặc thù số lượng thao tác Đọc (Xem danh sách, xem chi tiết, lọc) lớn gấp nhiều lần thao tác Ghi (Đặt hàng, cập nhật kho). ReadWriteLock khắc phục nhược điểm của Mutex bằng cách phân tách quyền truy cập:
   * **Khóa Đọc (Read Lock - Shared):** Cho phép nhiều luồng cùng truy cập HashMap để đọc dữ liệu cùng lúc mà không chặn lẫn nhau. Điều này tận dụng tối đa khả năng xử lý đa luồng của CPU.
   * **Khóa Ghi (Write Lock - Exclusive):** Mang tính độc quyền. Khi một luồng cần thay đổi dữ liệu (Ghi), nó sẽ yêu cầu khóa Ghi. Lúc này, hệ thống sẽ đợi các luồng Đọc hiện tại hoàn tất, sau đó chặn tất cả các truy cập mới cho đến khi luồng Ghi hoàn thành việc cập nhật Map và lưu xuống tệp.

**Kết luận:** Giải pháp ReadWriteLock toàn cục đảm bảo tính đúng đắn cho thao tác I/O tệp tin và cấu trúc Map, đồng thời giải phóng hiệu năng cho các tác vụ tra cứu dữ liệu

****

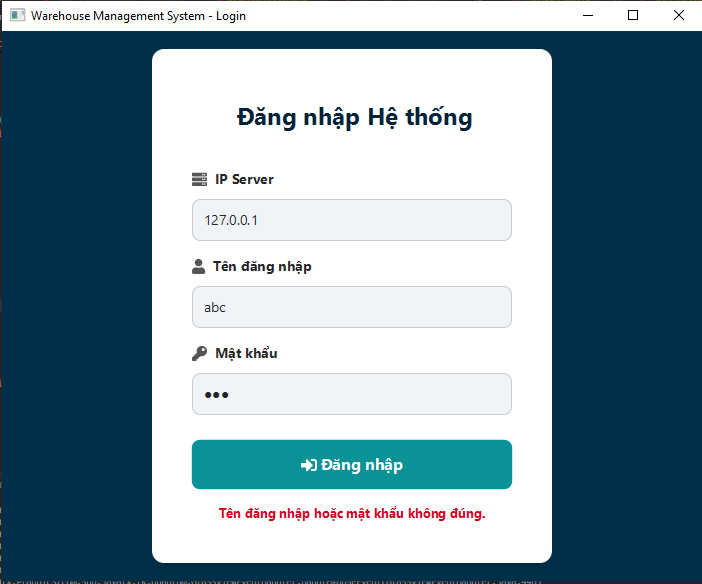
Hình 18 - Sơ đồ tuần tự mô tả 2 giao dịch diễn ra cùng lúc và cách khóa hoạt động

## 4. Giao diện

### 4.1. Trang Đăng nhập

Trang Đăng nhập là giao diện đầu tiên mà người dùng nhìn thấy khi khởi động ứng dụng.

* **Chức năng chính:**
  + Xác thực người dùng (Staff hoặc Manager).
  + Thiết lập phiên làm việc (Session) bằng cách gửi yêu cầu gRPC chứa thông tin đăng nhập đến Server.
* **Các Thành phần:**
  + Trường nhập liệu **IP Server**: Cho phép người dùng nhập địa chỉ IP của máy chủ gRPC.
  + Trường nhập liệu **Tên đăng nhập** (Username Field).
  + Trường nhập liệu **Mật khẩu** (Password Field).
  + Nút **Đăng nhập** (Login Button).
  + Khu vực hiển thị **thông báo lỗi** (ví dụ: tên đăng nhập hoặc mật khẩu không đúng, hoặc không thể kết nối Server).

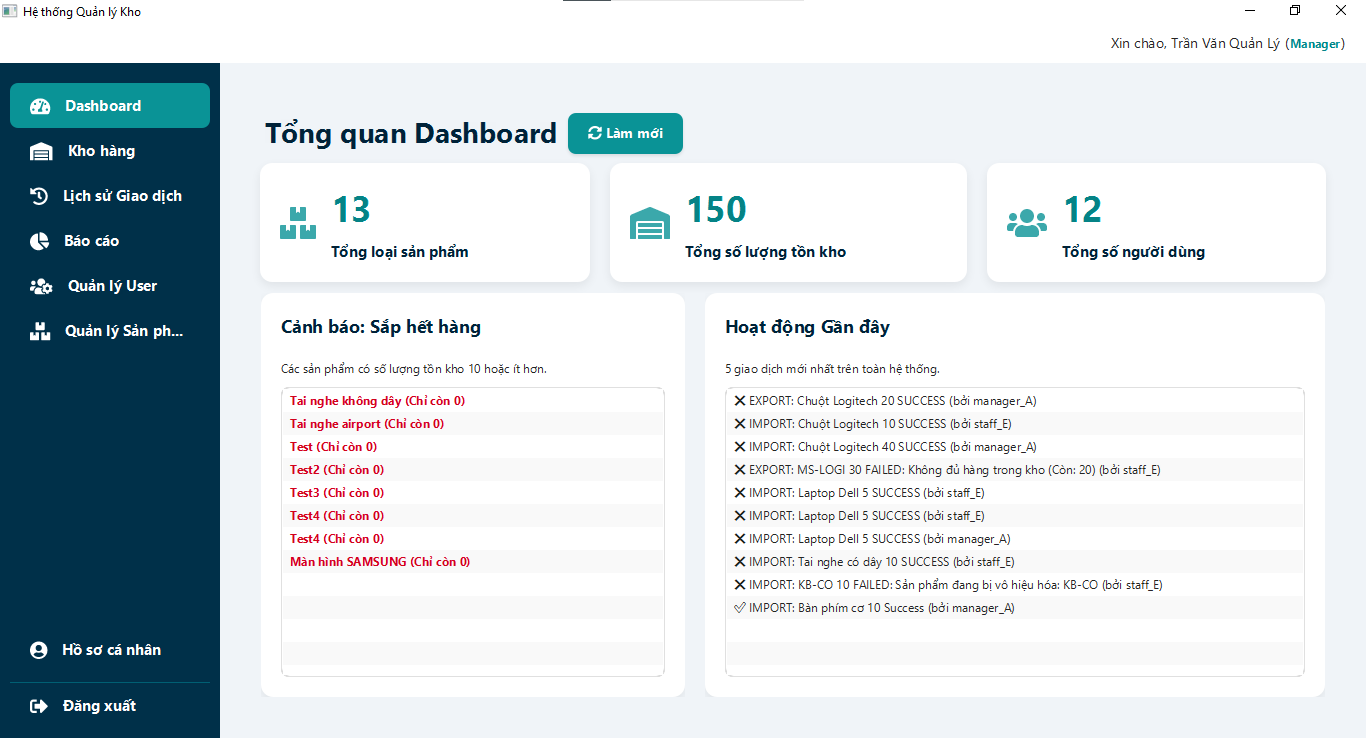


Hình 19 - Giao diện Trang Đăng nhập

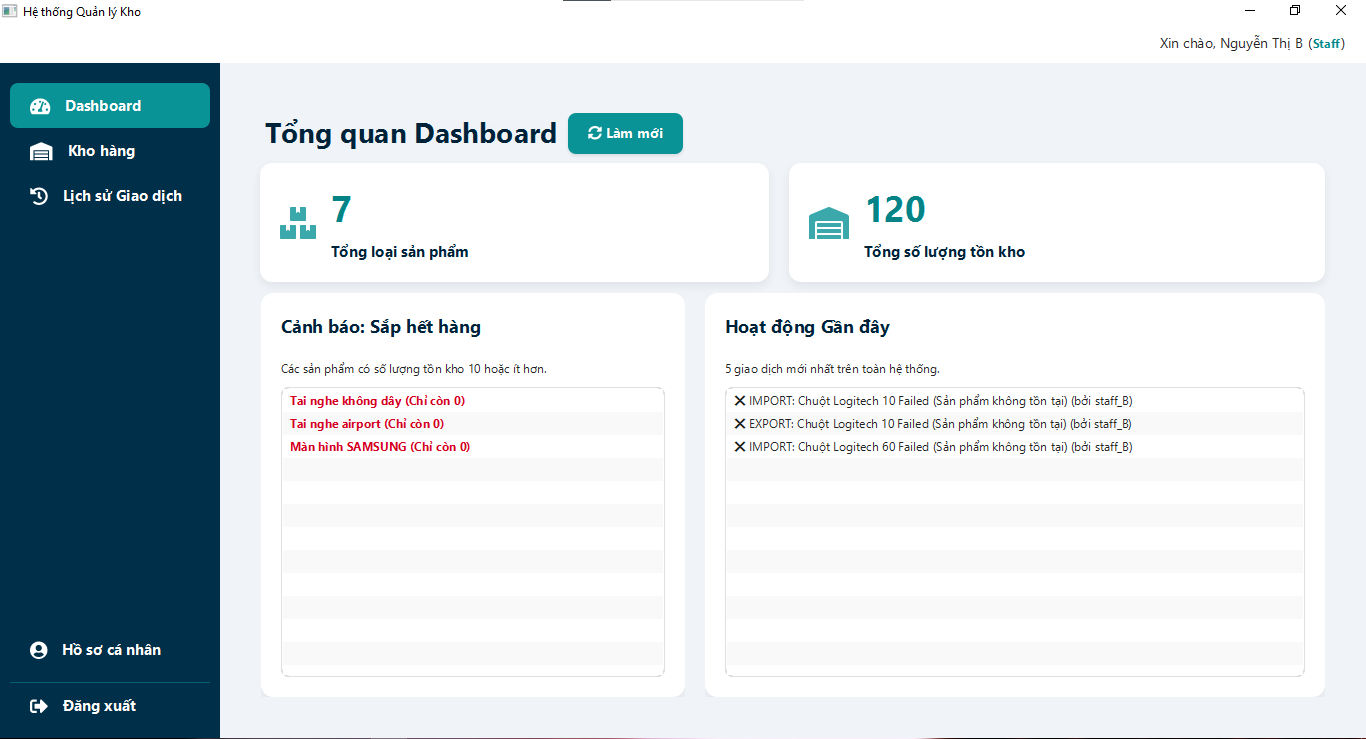
### 4.2. Trang Tổng quan

Đây là trang mặc định sau khi đăng nhập thành công, cung cấp cái nhìn tổng quát về tình hình kho hàng.

* **Quyền truy cập:** Cả **Staff** và **Manager** đều có thể truy cập.
* **Mô tả:** Màn hình cung cấp cái nhìn tổng quát về tình hình kho hàng thông qua các chỉ số hiệu suất chính (KPIs), cảnh báo sắp hết hàng và các log giao dịch gần đây.
* **Các Thành phần:**
  + Các thẻ (Card) hiển thị số liệu chính (ví dụ: **Tổng số sản phẩm**, **Tổng số giao dịch**, **Giá trị hàng tồn kho ước tính**).
  + **Biểu đồ (Chart):** Hiển thị xu hướng nhập/xuất hàng trong một khoảng thời gian.
  + **Bảng Tóm tắt Tồn kho:** Liệt kê các sản phẩm có số lượng tồn kho thấp hoặc biến động nhiều nhất.
* **Điểm khác biệt (Nghiệp vụ):** Nội dung hiển thị của Manager là tổng hợp dữ liệu **toàn bộ hệ thống**, trong khi Staff chỉ thấy dữ liệu **liên quan đến các giao dịch của họ** và không hiển thị **Tổng số người dùng** của hệ thống.



Hình 20 - Giao diện Dashboard của Manager

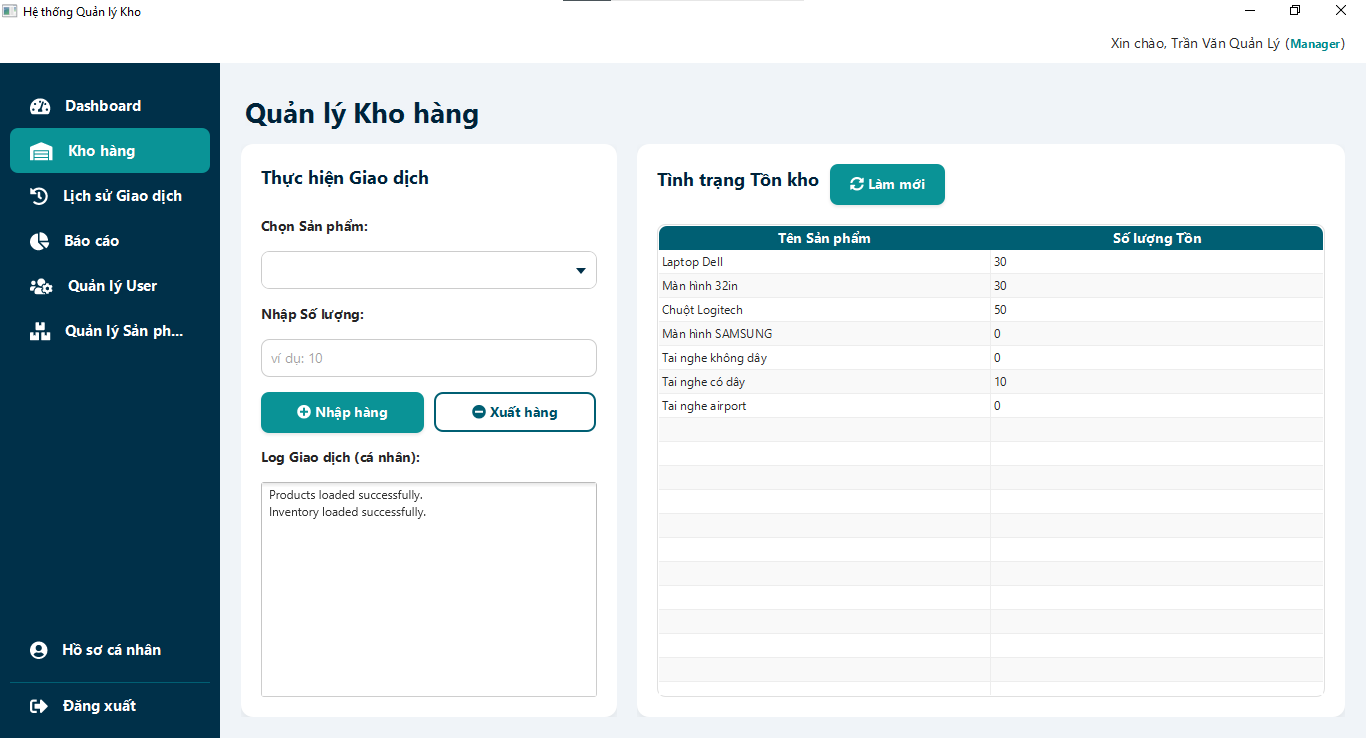


Hình 21 - Giao diện Dashboard của Staff

### 4.3. Trang Quản lý Kho

Đây là giao diện cốt lõi để thực hiện các nghiệp vụ nhập xuất hàng hóa.

* **Quyền truy cập:** Cả **Staff** và **Manager** đều có thể thực hiện nghiệp vụ này.
* **Mô tả:** Giao diện được chia thành khu vực hiển thị tồn kho, và khu vực nhập/xuất giao dịch.
* **Các Thành phần:**
  + **Bảng Tồn kho:** Hiển thị danh sách tất cả sản phẩm và số lượng tồn kho hiện tại (được truy vấn theo thời gian thực).
  + **Khu vực Giao dịch:** Chứa log giao dịch cá nhân các trường nhập liệu và các nút:
    - Tên Sản phẩm.
    - Số lượng.
    - Loại giao dịch: **Nhập hàng (Import)** (tăng tồn kho) hoặc **Xuất hàng (Export)** (giảm tồn kho).
  + Khi người dùng xác nhận giao dịch, Client gửi yêu cầu gRPC đến Server, nơi logic tương tranh (**ReadWriteLock** trên DataSource) đảm bảo tính toàn vẹn của số lượng tồn kho.

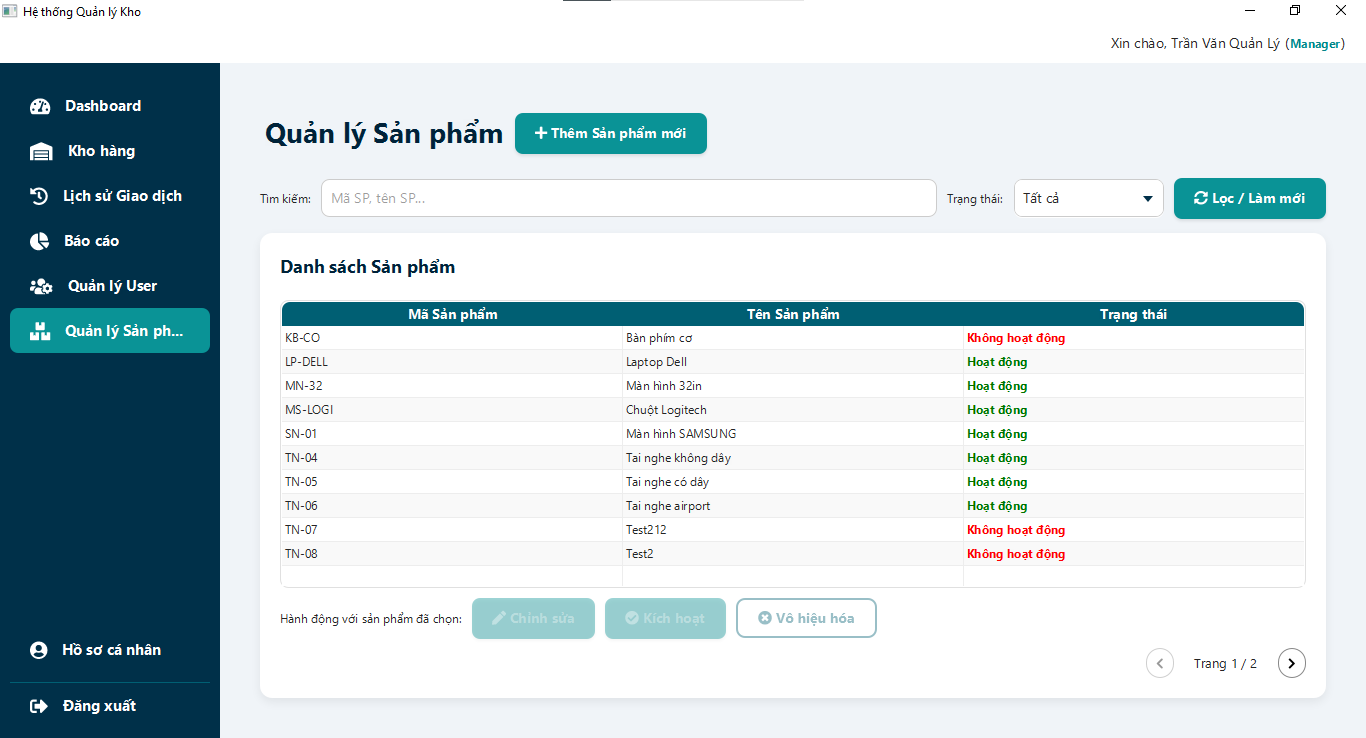


Hình 22 - Giao diện Quản lý Kho hàng

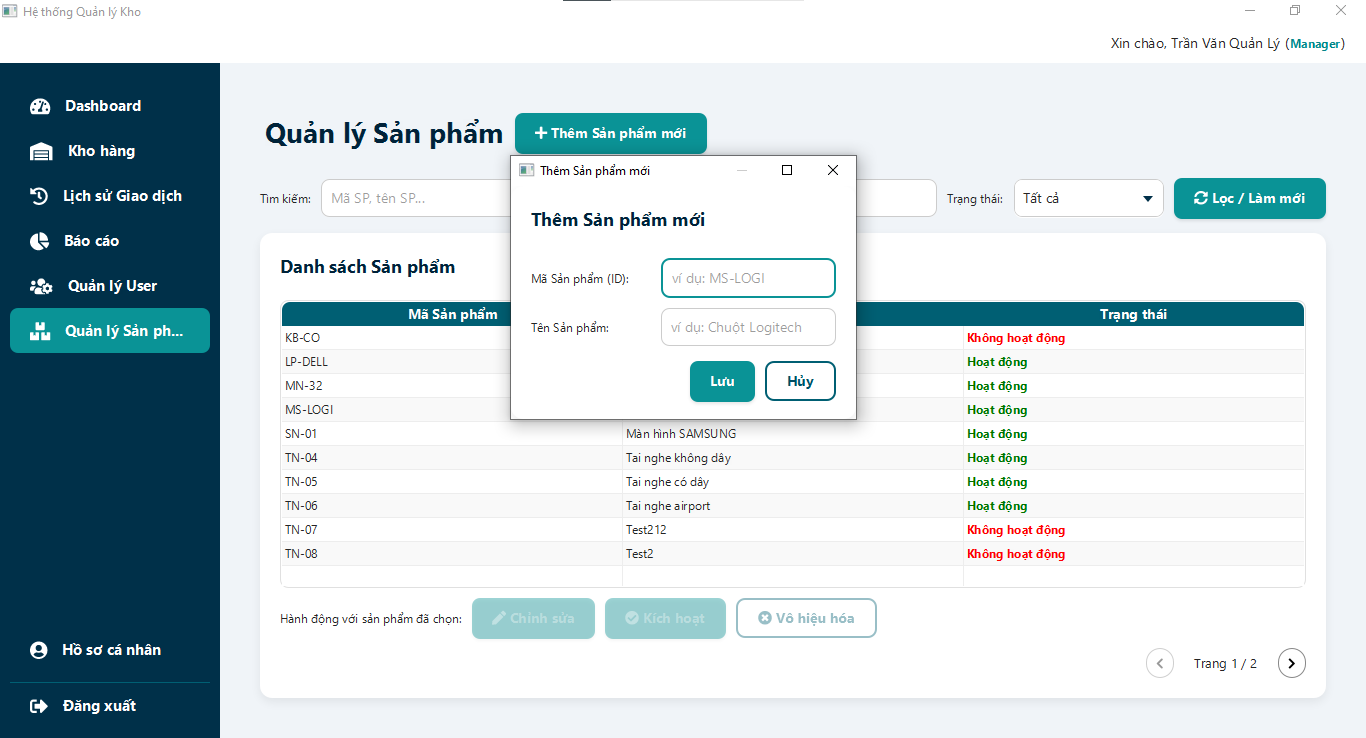
### 4.4. Trang Quản lý Sản phẩm

Đây là trang dành riêng cho việc quản lý danh mục sản phẩm.

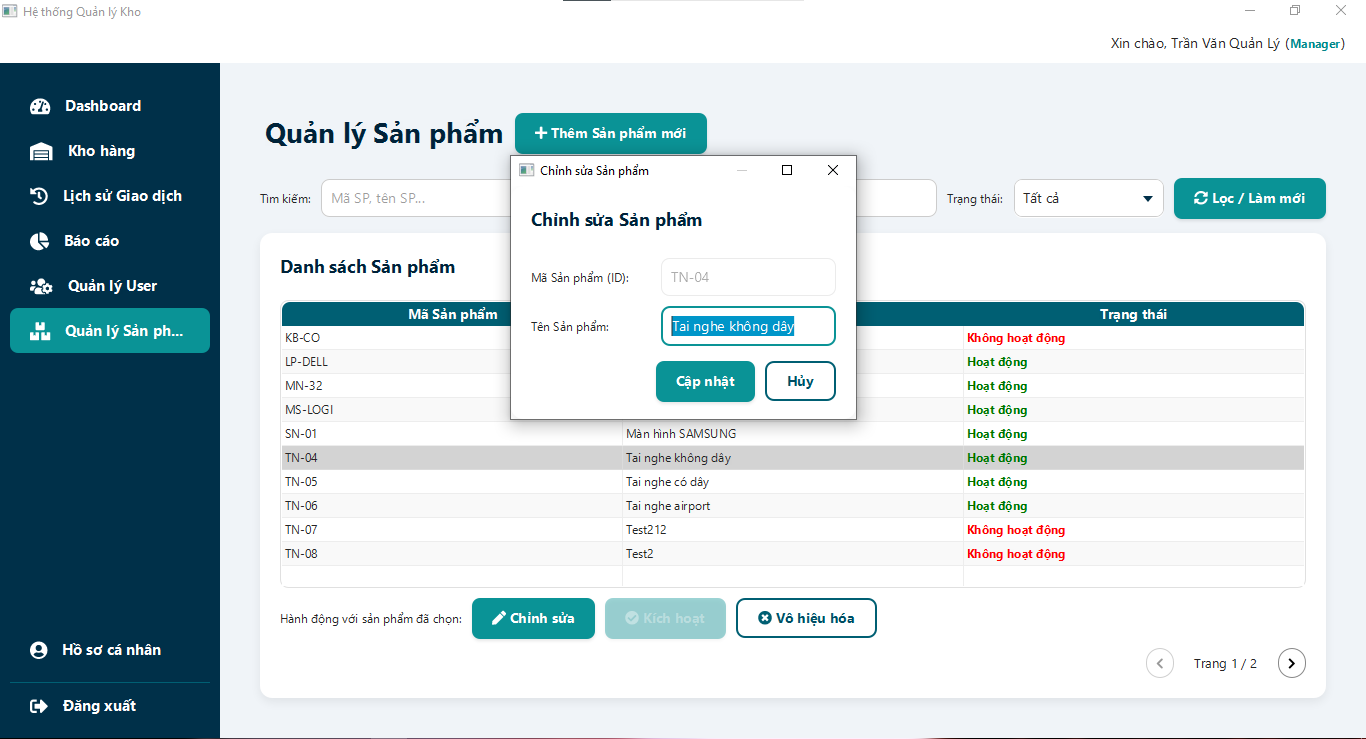
* **Quyền truy cập:** Chỉ dành cho **Manager**.
* **Mô tả:** Giao diện quản lý cho phép thực hiện các thao tác CRUD (Create, Read, Update, Delete) đối với sản phẩm.
* **Các Thành phần:**
  + **Bảng Sản phẩm:** Hiển thị chi tiết (Mã sản phẩm, Tên sản phẩm, Trạng thái) của tất cả sản phẩm.
  + **Thao tác CRUD:**
    - Nút **Thêm Sản phẩm**.
    - Nút **Chỉnh sửa** (khi chọn một sản phẩm).
    - Nút **Kích hoạt** và **Vô hiệu hóa** (khi chọn một sản phẩm).
  + Hộp thoại **Thêm Sản phẩm mới**:
    - Các trường nhập liệu: **Mã sản phẩm** và **Tên sản phẩm**.
    - Các nút thao tác **Lưu** và **Hủy**.
  + Hộp thoại **Chỉnh sửa Sản phẩm**:
    - Các trường nhập liệu: **Mã sản phẩm** (chỉ hiển thị không nhập)và **Tên sản phẩm**
    - Các nút thao tác **Cập nhật** và **Hủy**.



Hình 23 - Giao diện trang Quản lý Sản phẩm



Hình 24 - Giao diện hộp thoại Thêm Sản phẩm mới

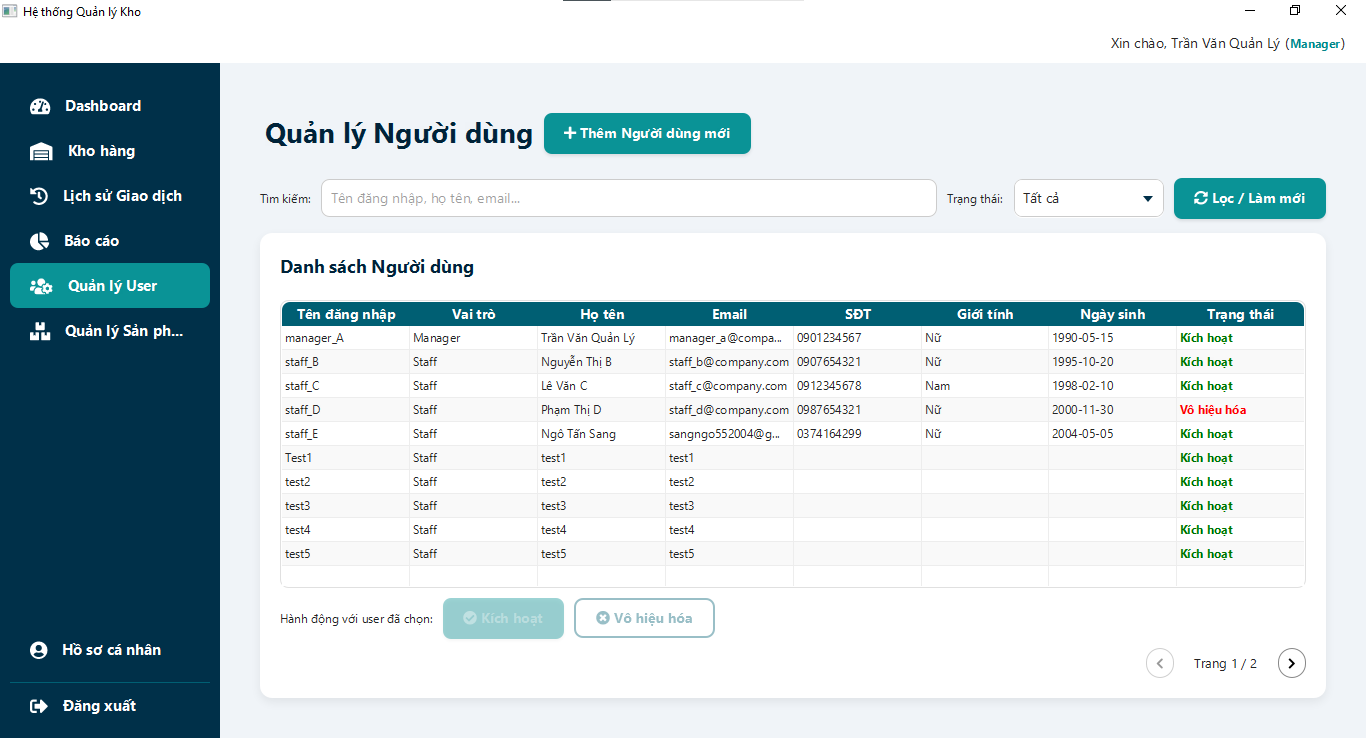


Hình 25 - Giao diện hộp thoại Chỉnh sửa Sản phẩm

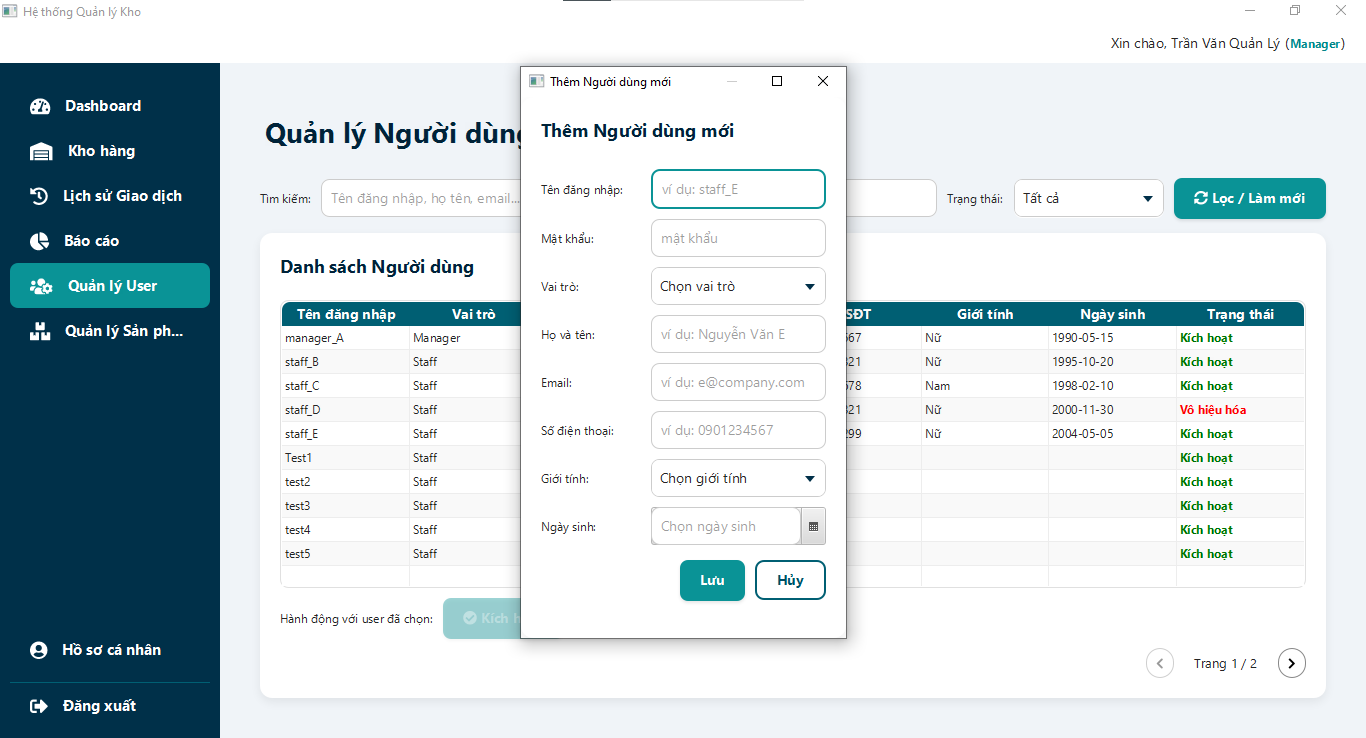
### 4.5. Trang Quản lý Người dùng

Đây là trang quản trị hệ thống, nơi Manager quản lý các tài khoản nhân viên.

* **Quyền truy cập:** Chỉ dành cho **Manager**.
* **Mô tả:** Giao diện cho phép quản lý thông tin và trạng thái của các tài khoản trong hệ thống.
* **Các Thành phần:**
  + **Bộ lọc/Tìm kiếm:** Cho phép tìm kiếm theo thông tin người dùng, lọc trạng thái.
  + **Bảng Người dùng:** Hiển thị danh sách tất cả tài khoản (**Staff** và **Manager**), bao gồm Tên đăng nhập, Tên, Email, Vai trò và Trạng thái.
  + **Thao tác Quản lý Tài khoản:**
    - Nút **Thêm Người dùng mới**.
    - Các tùy chọn Kích hoạt/Vô hiệu hóa tài khoản.
  + **Hộp thoại Thêm Người dùng mới:**
    - Các trường nhập liệu thông tin cá nhân (Tên đăng nhập, mật khẩu, email, số điện thoại,...).
    - Các nút **Lưu** và **Hủy**.



Hình 26 - Giao diện Quản lý người dùng

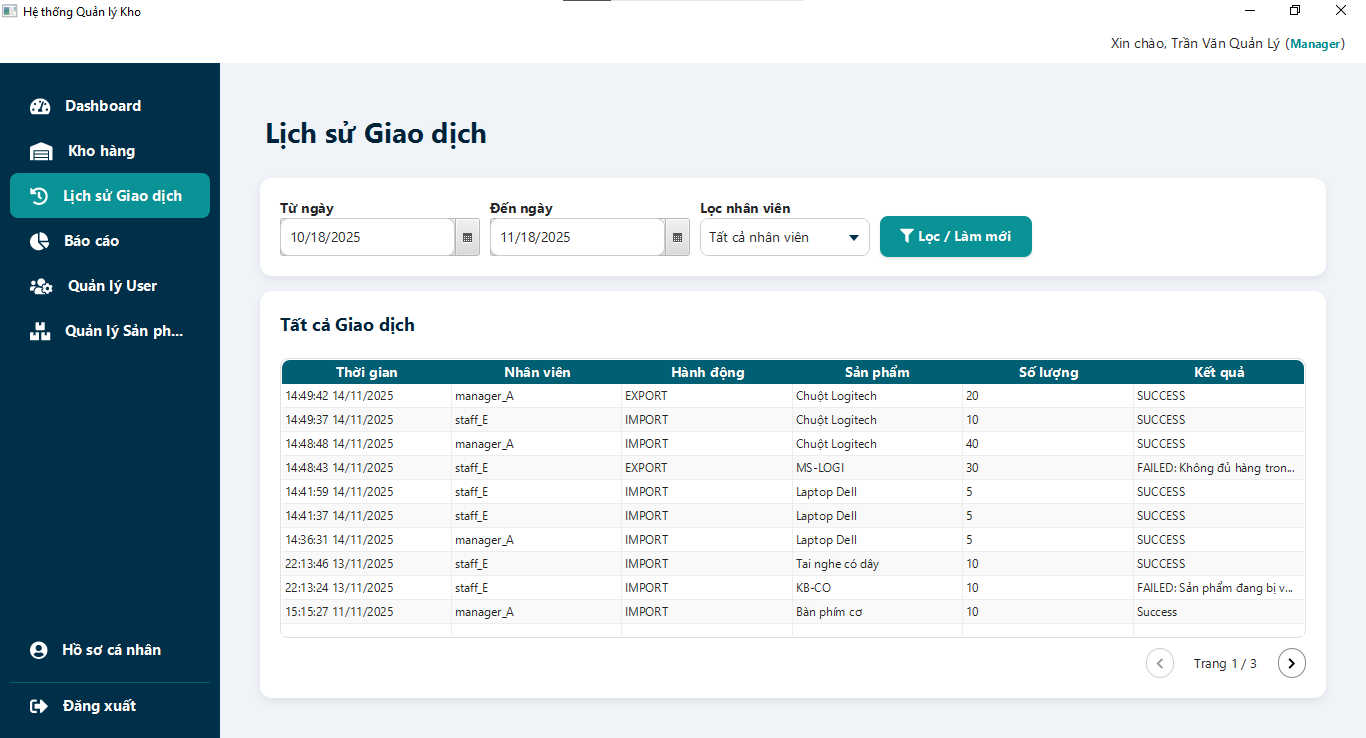


Hình 27 - Giao diện hộp thoại Thêm người dùng mới

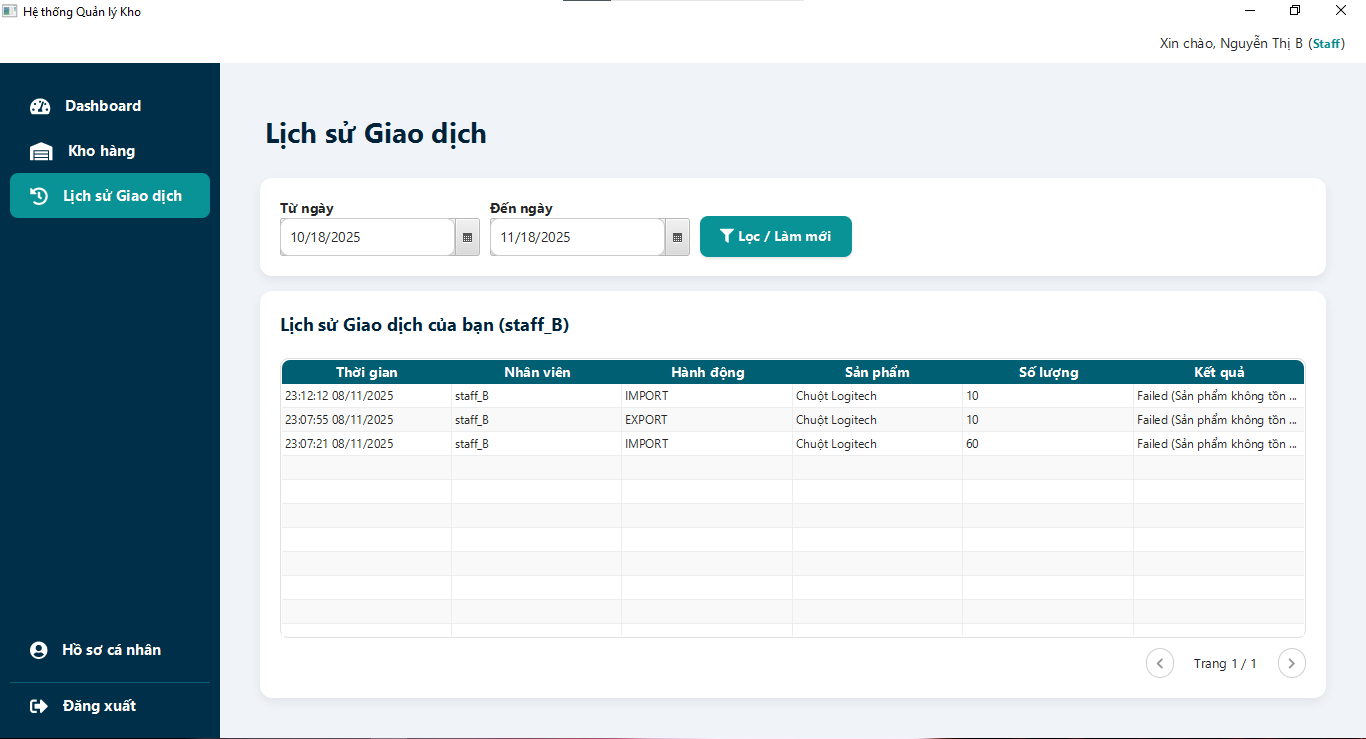
### 4.6. Trang Lịch sử Giao dịch

Trang hiển thị chi tiết các bản ghi giao dịch (Transaction).

* **Quyền truy cập:**
  + **Manager:** Xem lịch sử giao dịch của **toàn bộ** hệ thống.
  + **Staff:** Chỉ được phép xem lịch sử giao dịch của **chính bản thân** mình.
* **Mô tả:** Giao diện hiển thị bảng chi tiết các giao dịch đã xảy ra.
* **Các Thành phần:**
  + **Bảng Lịch sử Giao dịch:** Chứa các cột dữ liệu quan trọng như: Thời gian thực hiện, Người thực hiện, Hành động, Sản phẩm, Số lượng, Kết quả.
  + **Bộ lọc/Tìm kiếm:** Cho phép lọc theo ngày, loại giao dịch hoặc nhân viên (dành cho Manager).



Hình 28 - Giao diện Lịch sử Giao dịch của Manager

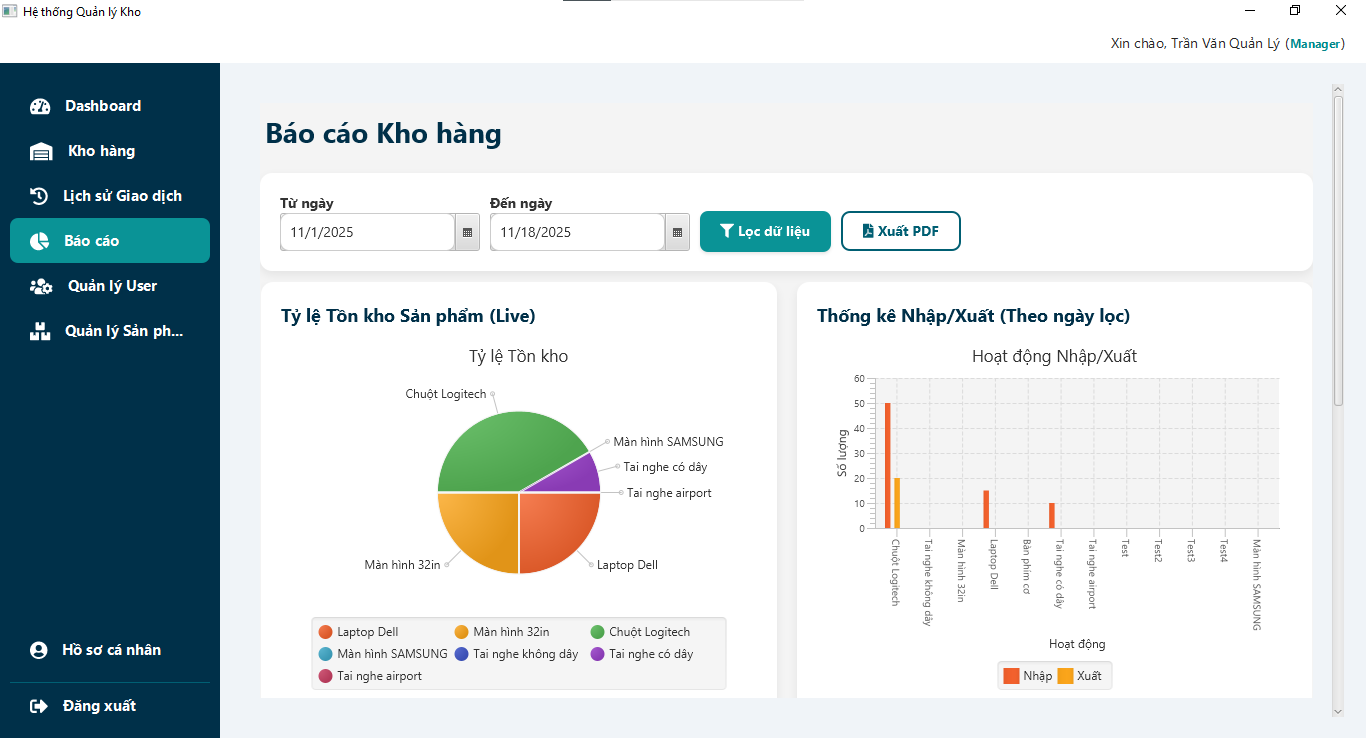


Hình 29 - Giao diện Lịch sử Giao dịch của Staff

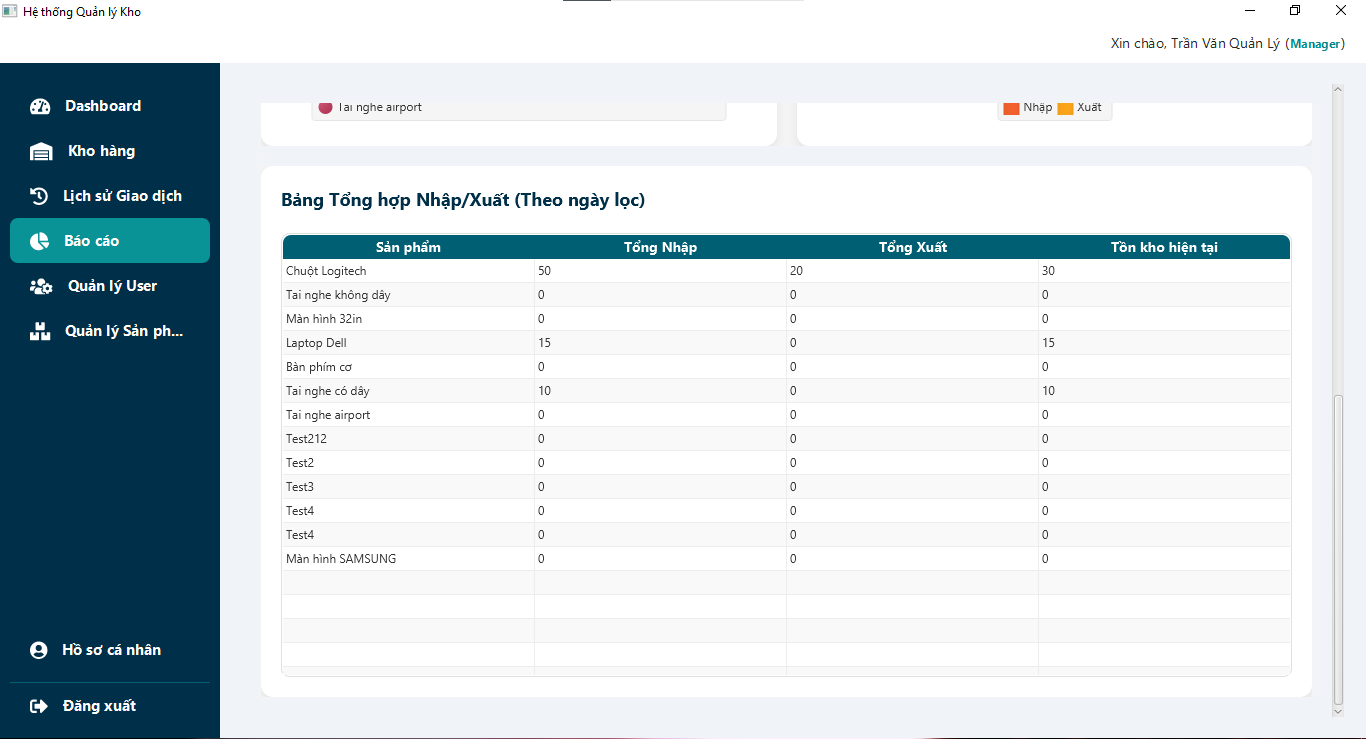
### 4.7. Trang Báo cáo

Trang tổng hợp và xuất dữ liệu báo cáo.

* **Quyền truy cập:** Chủ yếu dành cho **Manager** để xem báo cáo tổng hợp.
* **Mô tả:** Giao diện cho phép người dùng tạo và xem các báo cáo tổng hợp theo giai đoạn.
* **Các Thành phần:**
  + **Khu vực Điều khiển Báo cáo:** Lựa chọn phạm vi ngày báo cáo.
  + **Hiển thị Dữ liệu:** Hiển thị kết quả báo cáo dưới dạng bảng và biểu đồ.
  + **Chức năng Xuất PDF:** Nút cho phép xuất báo cáo ra các định dạng PDF.



Hình 30 - Giao diện Báo cáo - 1

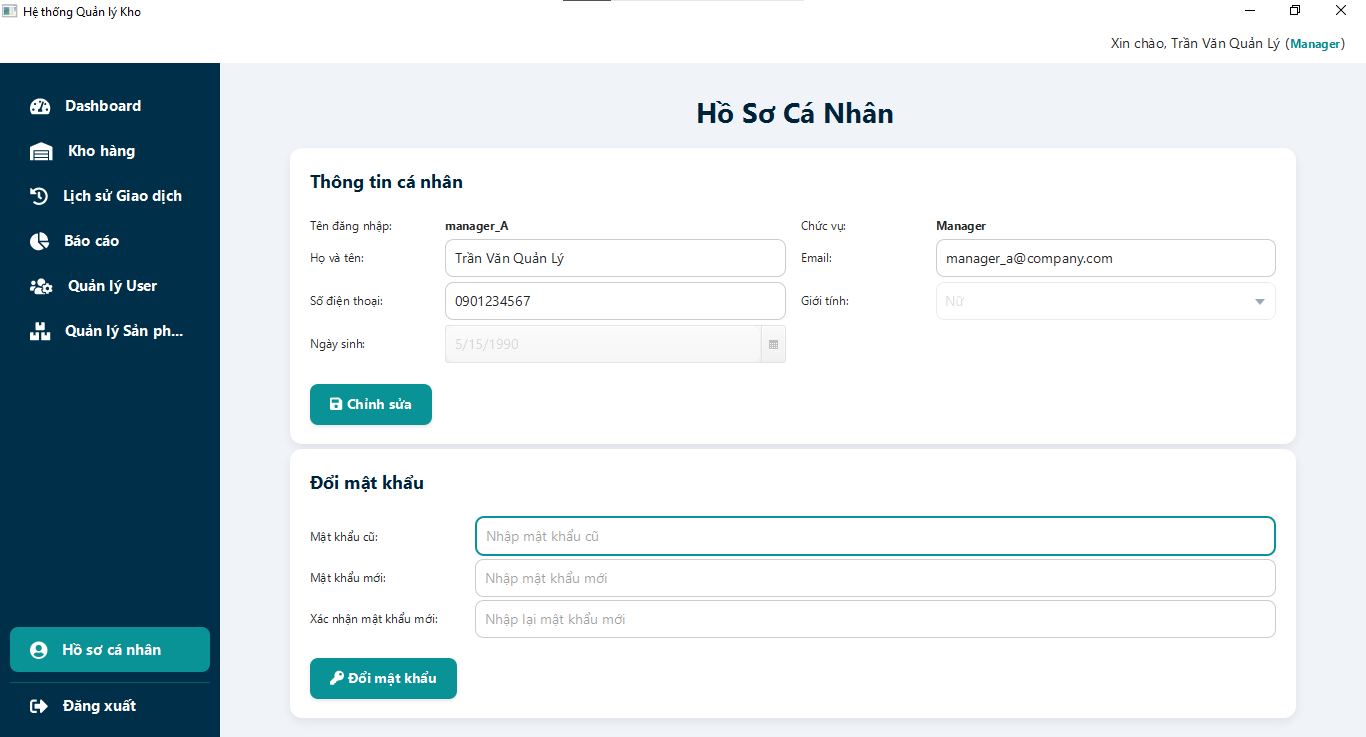


Hình 31 - Giao diện Báo cáo - 2

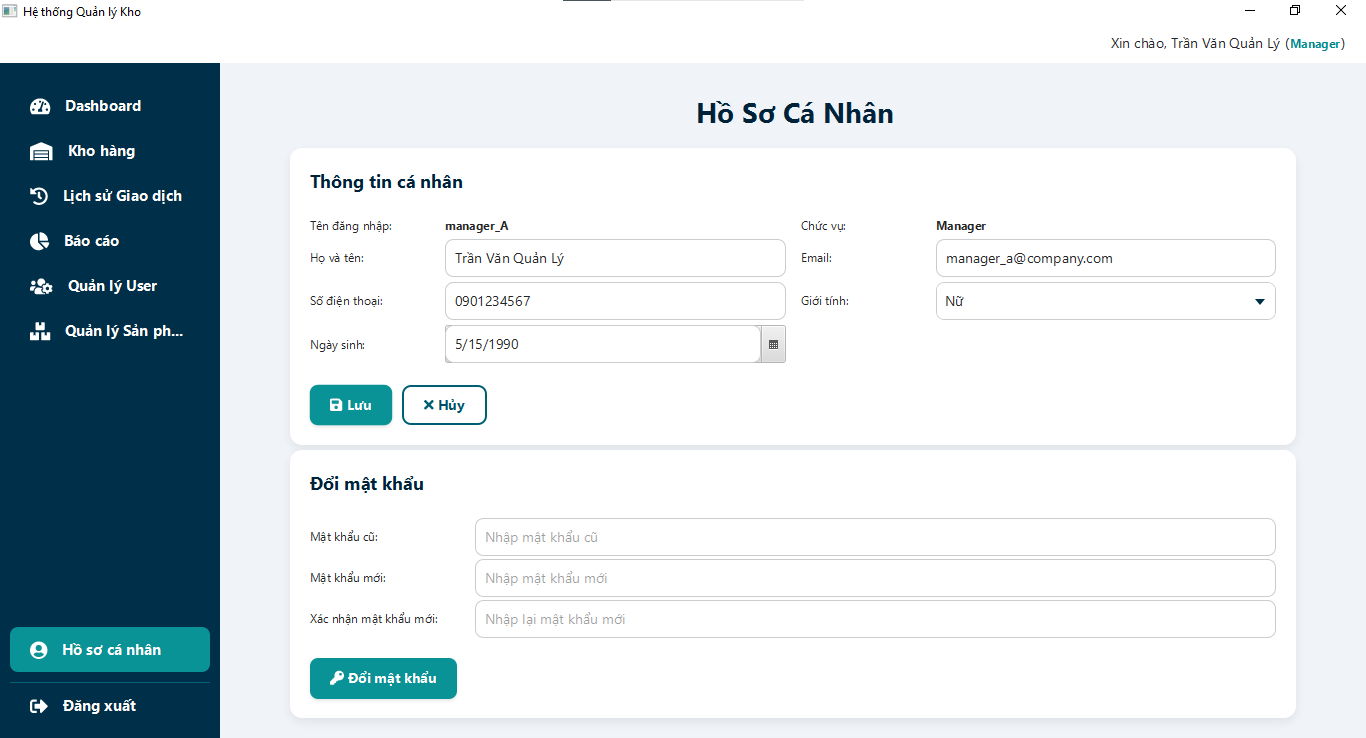
### 4.8. Trang Hồ sơ Cá nhân

Trang này cho phép người dùng xem và quản lý thông tin cá nhân của mình, đồng thời cung cấp chức năng đổi mật khẩu.

* **Quyền truy cập:** Cả **Staff** và **Manager** đều có thể truy cập.
* **Mô tả:** Giao diện được thiết kế rõ ràng, chia thành hai phần chính: **Thông tin Cá nhân** và **Đổi mật khẩu**.
* **Các Thành phần:**
  + **Phần Thông tin Cá nhân:** Hiển thị các trường thông tin cơ bản của người dùng, bao gồm:
    - Tên đăng nhập (Username).
    - Họ và tên.
    - Chức vụ/Vai trò (Staff hoặc Manager).
    - Số điện thoại, Email, Giới tính, Ngày sinh.
    - Nút **Chỉnh sửa** (cho phép người dùng cập nhật thông tin cá nhân).
  + **Phần Đổi mật khẩu:** Gồm các trường nhập liệu độc lập:
    - Mật khẩu cũ.
    - Mật khẩu mới.
    - Xác nhận mật khẩu mới.
    - Nút **Đổi mật khẩu**.



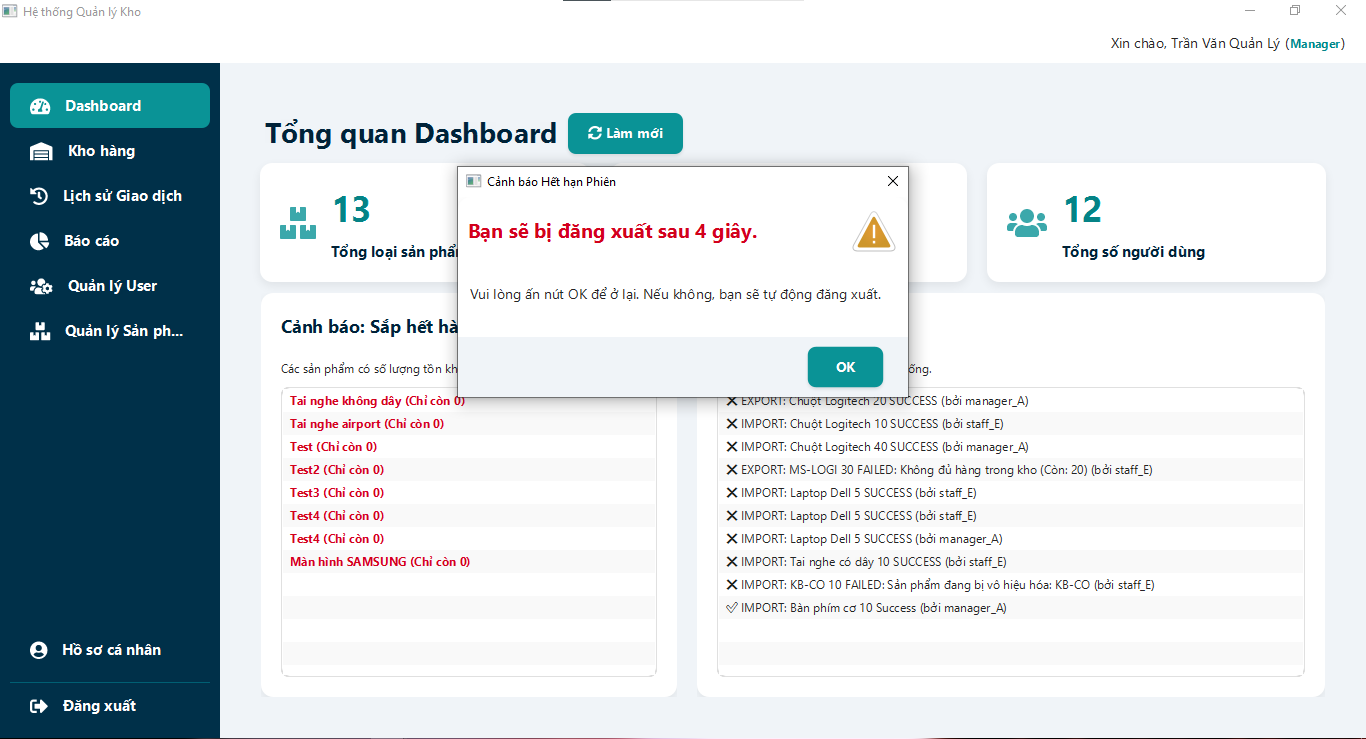
Hình 32 - Giao diện Hồ sơ cá nhân



Hình 33 - Giao diện Chỉnh sửa Thông tin cá nhân

### 4.9. Thông báo Hết hạn Phiên

* **Bối cảnh xuất hiện:** Thông báo này xuất hiện khi phiên làm việc của người dùng (JWT token) hết hạn do người dùng không hoạt động trong một khoảng thời gian nhất định (ví dụ: 4 giây trong ảnh minh họa). Đây là cơ chế bảo mật của hệ thống phân tán, được xử lý bởi logic phiên ở Client và việc kiểm tra JWT token ở Server.
* **Mô tả:** Thông báo là một hộp thoại (Dialog Box) nhỏ, nằm đè lên giao diện chính của ứng dụng (ví dụ: màn hình Dashboard).
  + **Tiêu đề:** "Cảnh báo Hết Hạn Phiên".
  + **Nội dung cảnh báo:** Hiển thị thông điệp rõ ràng như: "Bạn bị đăng xuất sau 4 giây.".
  + **Hướng dẫn:** Kèm theo hướng dẫn cho người dùng: "Vui lòng ấn nút OK để ở lại. Nếu không, bạn sẽ tự động đăng xuất."
  + **Nút hành động:** Chỉ có một nút **OK** để người dùng xác nhận.
* **Chức năng:** Hộp thoại này cảnh báo người dùng về việc sắp bị đăng xuất tự động để ngăn chặn truy cập trái phép khi người dùng không có mặt tại máy. Nếu người dùng không kịp nhấn **OK** trong thời gian đếm ngược, ứng dụng sẽ tự động chuyển về trang Đăng nhập và hủy bỏ phiên làm việc hiện tại.



Hình 34 - Giao diện Thông báo hết hạn phiên đăng nhập

# CHƯƠNG IV. KẾT LUẬN

Đồ án môn học "Các hệ thống phân tán" với đề tài "Xây dựng Ứng dụng tương tranh mô phỏng xử lý hàng tồn kho cho nhập xuất hàng hóa" đã được hoàn thành, đáp ứng các mục tiêu cốt lõi đã đề ra.

## 1. Kết quả đạt được

Nhóm đã xây dựng thành công một hệ thống Quản lý Kho (WMS) hoàn chỉnh, hoạt động theo mô hình Client-Server phân tán. Các kết quả chính bao gồm:

* **Về Kiến trúc:** Đã thiết kế và triển khai một kiến trúc N-Lớp (N-tier) rõ ràng phía Server , áp dụng thành công các nguyên lý Inversion of Control (IoC) và Dependency Injection (DI) để tạo ra một hệ thống có tính gắn kết lỏng (decoupled), dễ bảo trì và mở rộng.
* **Về Giao tiếp:** Tận dụng gRPC (sử dụng Protocol Buffers và HTTP/2) làm kênh giao tiếp duy nhất giữa Client và Server , đảm bảo hiệu suất cao và một "hợp đồng" dịch vụ (service contract) được định nghĩa chặt chẽ.
* **Về Nghiệp vụ:** Hệ thống đã mô phỏng đầy đủ các nghiệp vụ kho hàng cốt lõi, bao gồm quản lý sản phẩm , nhập/xuất kho , và xem báo cáo.
* **Về Bảo mật & Phân quyền:** Đã triển khai thành công cơ chế xác thực bằng JWT và BCrypt , kết hợp với Interceptor để thực thi việc Kiểm soát Truy cập Dựa trên Vai trò (RBAC) một cách hiệu quả, phân định rõ ràng quyền hạn giữa Manager và Staff.
* **Về Giao diện:** Xây dựng một ứng dụng Client trực quan bằng JavaFX , tách biệt rõ ràng giao diện (FXML), logic (Controller) và phong cách (CSS).
* **Về Thách thức Cốt lõi (Tương tranh):** Đây là thành tựu quan trọng nhất của đồ án. Bằng cách chọn lưu trữ bằng file JSON để làm nổi bật vấn đề, nhóm đã xác định chính xác rủi ro "Mất Cập nhật" (Lost Update) . Nhóm đã thiết kế và triển khai thành công giải pháp khóa cấp ứng dụng (application-level lock) bằng ReadWriteLock tại lớp DataSource. Giải pháp này giải quyết triệt để vấn đề toàn vẹn dữ liệu trong khi vẫn tối ưu hóa hiệu năng cho kịch bản "Đọc nhiều, Ghi ít" (Read-many, Write-few) của hệ thống.

## 2. Hạn chế và Tồn tại

Bên cạnh những kết quả tích cực, đồ án cũng bộc lộ một số hạn chế mang tính thiết kế:

* **Hiệu năng Lưu trữ:** Quyết định sử dụng file JSON tuy giúp đơn giản hóa việc cài đặt và làm rõ bài toán tương tranh, nhưng đây chính là điểm yếu lớn nhất về hiệu năng và khả năng mở rộng.
* **Khóa Toàn cục (Global Lock):** Giải pháp ReadWriteLock là một cơ chế khóa "thô" (coarse-grained). Mọi thao tác Ghi (thêm, sửa, xóa) đều yêu cầu một writeLock độc quyền, khiến tất cả các nghiệp vụ ghi phải thực hiện một cách tuần tự. Hệ thống không hỗ trợ khóa ở mức độ bản ghi (row-level locking) như các CSDL chuyên dụng.
* **Thao tác "Đọc-Sửa-Ghi":** Mỗi nghiệp vụ Ghi đòi hỏi phải đọc toàn bộ file JSON vào bộ nhớ, sửa đổi, sau đó ghi đè lại toàn bộ tệp . Thao tác này sẽ trở nên cực kỳ chậm chạp và tốn tài nguyên khi lượng dữ liệu (sản phẩm, giao dịch) tăng lên.

## 3. Hướng phát triển Tương lai

Từ những hạn chế trên, nhóm đề xuất các hướng phát triển để nâng cấp hệ thống:

* **Thay thế Lớp Lưu trữ:** Hướng ưu tiên hàng đầu là thay thế các DataSource sử dụng file JSON bằng một Hệ Quản trị Cơ sở dữ liệu (CSDL) thực thụ (như PostgreSQL, MySQL). Nhờ kiến trúc IoC/DI đã thiết kế , việc thay đổi này chỉ cần tập trung vào việc triển khai lại các lớp trong repository và datasource mà không ảnh hưởng đến service hay grpc. Điều này sẽ giải quyết triệt để các vấn đề về hiệu năng và khóa tương tranh.
* **Áp dụng gRPC Streaming:** Tận dụng các tính năng nâng cao của gRPC như Server Streaming RPC. Ví dụ, Server có thể "stream" (phát) các cập nhật về số lượng tồn kho theo thời gian thực đến tất cả các Client đang kết nối, thay vì Client phải bấm nút "Làm mới" để lấy dữ liệu mới.
* **Mở rộng Báo cáo:** Với nền tảng CSDL, mô-đun báo cáo có thể được mở rộng để thực hiện các truy vấn phân tích phức tạp hơn.
* **Triển khai và Giám sát:** Container hóa (Dockerize) ứng dụng Server và triển khai lên một nền tảng điều phối (như Kubernetes ) để tận dụng tối đa lợi thế của kiến trúc microservices mà gRPC được thiết kế để hỗ trợ.

Nhìn chung, đồ án đã là một bài tập thực tế giá trị, giúp nhóm áp dụng thành công các lý thuyết phức tạp của hệ thống phân tán, xử lý tương tranh, và các mẫu thiết kế phần mềm hiện đại vào một ứng dụng nghiệp vụ cụ thể.

# TÀI LIỆU THAM KHẢO

[1] Google LLC. (2025). *gRPC Documentation*. [Trực tuyến]. Có sẵn:<https://grpc.io/docs/>

[2] Google LLC. (2025). *Protocol Buffers Documentation*. [Trực tuyến]. Có sẵn:<https://protobuf.dev/>

[3] Oracle Corporation. (2025). *JavaFX Documentation*. [Trực tuyến]. Có sẵn: [https://openjfx.io/documentation/](https://www.google.com/search?q=https://openjfx.io/documentation/)

[4] Oracle Corporation. (2025). *JDK 17 Documentation*. [Trực tuyến]. Có sẵn:<https://docs.oracle.com/en/java/javase/17/>

[5] Oracle Corporation. (2025). *Interface java.util.concurrent.locks.ReadWriteLock*. [Trực tuyến]. Có sẵn:<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/locks/ReadWriteLock.html>

[6] M. Fowler. (2005). *Inversion of Control*. [Trực tuyến]. Có sẵn:<https://martinfowler.com/bliki/InversionOfControl.html>

[7] M. Jones, J. Bradley, & N. Sakimura. (2015). *JSON Web Token (JWT) - RFC 7519*. Internet Engineering Task Force (IETF). [Trực tuyến]. Có sẵn:<https://tools.ietf.org/html/rfc7519>

[8] D. K. R. Ylonen. (2025). *jBCrypt - A Java implementation of the OpenBSD bcrypt password hashing algorithm*. [Phần mềm]. Có sẵn: [https://github.com/mindrot/jbcrypt](https://www.google.com/search?q=https://github.com/mindrot/jbcrypt)