**BỘ GIÁO DỤC VÀ ĐÀO TẠO**

**TRƯỜNG ĐẠI HỌC MỞ THÀNH PHỐ HỒ CHÍ MINH**

**NGUYỄN ANH KHOA**

**HỆ THỐNG GIAO LƯU MUA BÁN BẤT ĐỘNG SẢN**

**ĐỒ ÁN NGÀNH**

**NGÀNH CNTT**

**TP. HỒ CHÍ MINH, 2025BỘ GIÁO DỤC VÀ ĐÀO TẠO**

**TRƯỜNG ĐẠI HỌC MỞ THÀNH PHỐ HỒ CHÍ MINH**

**NGUYỄN ANH KHOA**

**HỆ THỐNG GIAO LƯU MUA BÁN BẤT ĐỘNG SẢN**

**Mã số sinh viên: 2251052052**

**ĐỒ ÁN NGÀNH**

**NGÀNH CNTT**

**Giảng viên hướng dẫn: THS. DƯƠNG HỮU THÀNH**

**TP. HỒ CHÍ MINH, 2025**

LỜI CẢM ƠN

Trong suốt quá trình học tại trường, em đã có cơ hội lĩnh hội được rất nhiều kiến thức hay và bổ ích từ các quý Thầy/Cô là giảng viên của khoa Công nghệ thông tin Trường Đại Học Mở Thành phố Hồ Chí Minh. Bằng sự tâm huyết và tấm lòng nhiệt thành mà Thầy/Cô đã kiên nhẫn trong suốt quá trình giảng dạy, em càng thêm trân trọng và biết ơn những tri thức quý giá cùng sự tận tâm mà Thầy/Cô đã dành cho chúng em..

Ngày hôm nay có cơ hội để thực hiện một dự án đặc biệt quan trọng như Đồ Án Ngành, em đã có cơ hội để ứng dụng các kiến thức mà mình đã học được vào một dự án đặc biệt quan trọng mang tính chuyển mình, em nhận thấy công ơn của Thầy cô là vô cùng lớn vì nếu không có sự dìu dắt từ quý thầy cô, chắc có lẽ sẽ không có em của ngày hôm nay.

Để hoàn thành đồ án, em không thể nào quên công ơn và sự giúp sức cực kỳ to lớn từ một người thầy mà em hết mực trân quý: ThS. Dương Hữu Thành. Thầy đã hỗ trợ em tận tình, tận tâm trong suốt quá trình em thực hiện Đồ Án, là người đã đưa ra những đề xuất, định hướng, những phương án khả thi để giải quyết các vấn đề gặp phải. Chính nhờ sự giúp sức đó, em đã kịp thời hoàn thành Báo cáo này đúng hạn và hoàn thiện kĩ lưỡng. Từ tận sâu trong đáy lòng, em xin gửi lời cảm ơn sâu sắc đến Thầy.

NHẬN XÉT CỦA GIÁO VIÊN HƯỚNG DẪN

TÓM TẮT ĐỐ ÁN NGÀNH

**Mục tiêu đề tài:**

Đồ án này hướng đến việc xây dựng một ứng dụng giao lưu mua bán bất động sản minh bạch, hiệu quả bằng cách phát triển nền tảng mobile app tích hợp tìm kiếm, phân tích dữ liệu và gợi ý thông minh. Người dùng có thể đăng tin, tìm kiếm bất động sản theo tiêu chí như vị trí, giá, diện tích, tiện ích,…

**Giải pháp thực hiện:**

Phát triển ứng dụng di động (Flutter) với giao diện hiện đại, tối ưu trải nghiệm.

Xây dựng trên nền tảng Django REST Framework mạnh mẽ cho phép người dùng đăng tải và quản lý thông tin bất động sản một cách chi tiết.

Tích hợp bản đồ số giúp người dùng trực quan hóa vị trí và khu vực xung quanh bất động sản.

Tích hợp các công cụ tương tác thời gian thực để kết nối người mua và người bán một cách hiệu quả.

Gợi ý phong thủy cá nhân hóa dựa trên năm sinh, bản mệnh người dùng.

Xây dựng diễn đàn cộng đồng: đánh giá tin đăng, cảnh báo rủi ro, chia sẻ kinh nghiệm.

Sử dụng kiến trúc bất đồng bộ để xử lý các tác vụ nền, từ đó giảm bớt gánh nặng xử lý nghiệp vụ trong luồng chính và nâng cao hiệu quả vận hành hệ thống.

**Kết quả đạt được:**  
Hệ thống giúp người dùng tiếp cận thông tin bất động sản minh bạch hơn, dễ dàng đưa ra quyết định mua bán nhờ các công cụ phân tích và tư vấn thông minh. Đồng thời, cộng đồng người dùng góp phần tạo nên môi trường đáng tin cậy, giảm thiểu rủi ro, nâng cao hiệu quả giao dịch.

**MỤC LỤC**

[LỜI CẢM ƠN 1](#_Toc208180353)

[NHẬN XÉT CỦA GIÁO VIÊN HƯỚNG DẪN 2](#_Toc208180354)

[TÓM TẮT ĐỐ ÁN NGÀNH 3](#_Toc208180355)

[DANH MỤC TỪ VIẾT TẮT 5](#_Toc208180356)

[DANH MỤC HÌNH VẼ 6](#_Toc208180357)

[DANH MỤC BẢNG 7](#_Toc208180358)

[MỞ ĐẦU 8](#_Toc208180359)

[TÀI LIỆU THAM KHẢO 56](#_Toc208180360)

[PHỤ LỤC 57](#_Toc208180361)

DANH MỤC TỪ VIẾT TẮT

DANH MỤC HÌNH VẼ

Không tìm thấy mục nhập nào của bảng hình minh họa.

DANH MỤC BẢNG

Không tìm thấy mục nhập nào của bảng hình minh họa.

MỞ ĐẦU

# TỔNG QUAN ĐỀ TÀI

## Giới thiệu đề tài

Thị trường bất động sản Việt Nam hiện nay đang phát triển mạnh mẽ, với nhu cầu mua bán, đầu tư ngày càng tăng cao tại ở cả khu vực thành thị và nông thôn. Tuy nhiên, bên cạnh tiềm năng, thị trường vẫn tồn tại nhiều **thách thức lớn,** trong đó nổi bật là tình trạng **thiếu minh bạch thông tin.**

Người mua thường gặp khó khăn trong việc xác thực nguồn gốc pháp lý, so sánh giá thị trường hay đánh giá tiềm năng đầu tư của một bất động sản. Trong khi đó, người bán cũng khó tiếp cận đúng đối tượng khách hàng mà mình muốn hướng tới. Nhiều trường hợp **lừa đảo, thổi giá, tin giả**, hoặc **thiếu công cụ hỗ trợ phân tích** khiến quá trình giao dịch trở nên rủi ro, tốn thời gian và chi phí.

Để góp phần giải quyết những vấn đề trên, nhóm đã thực hiện đề tài **“Hệ thống giao lưu mua bán bất động sản”,** với mong muốn xây dựng một nền tảng trực tuyến hỗ trợ người dùng đăng tin, tìm kiếm, phân tích và trao đổi thông tin một cách **hiệu quả, minh bạch và thông minh hơn**. Qua đó, đề tài không chỉ đáp ứng nhu cầu thực tế cấp thiết, mà còn hướng đến việc nâng cao chất lượng giao dịch, góp phần vào sự phát triển bền vững của thị trường bất động sản Việt Nam.

## Lý do chọn đề tài

Việc thiếu minh bạch thông tin trong thị trường bất động sản không chỉ gây trở ngại cho quá trình mua bán, mà còn dẫn đến rủi ro cao như mua phải tài sản không đúng giá trị, dính pháp lý, hoặc bị lừa đảo. Trong khi đó, nhu cầu tra cứu, so sánh và phân tích thông tin trước khi quyết định đầu tư là rất lớn nhưng chưa có nhiều nền tảng hỗ trợ hiệu quả, minh bạch và tin cậy.

Việc xây dựng hệ thống giao lưu mua bán bất động sản không chỉ giúp các bên tham gia dễ dàng tiếp cận thông tin chính xác, mà còn góp phần **nâng cao tính minh bạch của thị trường**, giảm thiểu rủi ro và tiết kiệm thời gian trong quá trình giao dịch. Hệ thống còn mở ra hướng phát triển các công nghệ mới như **AI, dữ liệu lớn và phân tích phong thủy**, mang lại giá trị thực tiễn cao, phù hợp với xu thế chuyển đổi số trong lĩnh vực bất động sản hiện nay.

## Mục tiêu đề tài

Xây dựng một ứng dụng mobile giao lưu mua bán bất động sản thân thiện, dễ sử dụng.

Cho phép người dùng đăng tin, tìm kiếm bất động sản theo các tiêu chí cụ thể (vị trí, diện tích, giá cả, tiện ích…).

Tích hợp chức năng phân tích, so sánh và tư vấn thông minh giúp người dùng đưa ra quyết định chính xác hơn.

Cung cấp tính năng gợi ý theo phong thủy dựa trên thông tin cá nhân như năm sinh, bản mệnh – yếu tố được nhiều người Á Đông nói chung và người Việt Nam quan tâm.

Xây dựng một mạng lưới người dùng để chia sẻ kinh nghiệm, cảnh báo rủi ro và đánh giá tin đăng.

## Phương pháp thực hiện

Trong quá trình thực hiện đề tài, nhóm đã tiến hành khảo sát một số hệ thống, nền tảng và mô hình đã có trong lĩnh vực giao dịch bất động sản để hiểu rõ cách vận hành, ưu điểm và hạn chế của từng phương pháp. Từ đó, nhóm lựa chọn các hướng tiếp cận phù hợp nhằm đảm bảo hệ thống đáp ứng tốt yêu cầu thực tế.

Cụ thể, nhóm thực hiện các bước như sau:

Khảo sát, phân tích hệ thống thực tế: Tìm hiểu một số trang web và ứng dụng bất động sản đang phổ biến hiện nay như Batdongsan.com.vn, Alonhadat, Nhatot, Guland... để đánh giá về giao diện, tính năng, khả năng tìm kiếm và mức độ minh bạch thông tin.

Thu thập dữ liệu: Tiến hành thu thập dữ liệu thực tế từ các nguồn công khai như bản đồ, thông tin quy hoạch, các tin rao bán thực, đồng thời tạo bộ dữ liệu giả lập để phục vụ quá trình thử nghiệm và mô phỏng.

Thử nghiệm mô hình và đánh giá: Áp dụng một số mô hình thiết kế hệ thống, từ đơn giản đến phức tạp (ví dụ: mô hình MVC, RESTful API) để so sánh hiệu quả hoạt động, tính mở rộng và độ linh hoạt. Qua đó, nhóm lựa chọn mô hình phù hợp nhất với nhu cầu triển khai thực tế.

Lựa chọn công nghệ và triển khai: Dựa trên kết quả đánh giá, nhóm quyết định sử dụng Django REST Framework để xây dựng phần backend API mạnh mẽ, dễ mở rộng và bảo mật tốt; đồng thời sử dụng Flutter để phát triển giao diện người dùng (frontend) linh hoạt, hỗ trợ đa nền tảng (web và mobile). Việc kết hợp hai công nghệ này giúp hệ thống dễ dàng triển khai và nâng cấp trong tương lai.

Thông qua phương pháp này, nhóm có thể xây dựng một sản phẩm vừa bền vững về kiến trúc, vừa đáp ứng yêu cầu về tính năng, hiệu suất và trải nghiệm người dùng trong môi trường thực tế.

## Bố cục báo cáo

Báo cáo đồ ngành này có 4 chương:

Chương 1. Tổng quan đề tài: Nhằm giới thiệu bối cảnh của đề tài, phương pháp nghiên cứu và mục tiêu của đề tài.

Chương 2. Cơ sở lý thuyết: Trình bày nền tảng lý thuyết áp dụng cho đề tài.

Chương 3. Kiến trúc hệ thống: Đi sâu vào việc phân tích các yêu cầu chức năng, xác định các tác nhân và mô hình hóa các luồng hoạt động của hệ thống thông qua biểu đồ Use Case. Chương này cũng trình bày chi tiết về thiết kế cơ sở dữ liệu và các mối quan hệ giữa các thực thể.

Chương 4. Kết luận và hướng phát triển: Tổng kết lại những kết quả đã đạt được, đánh giá các ưu điểm và những mặt còn hạn chế của dự án. Đồng thời, đề xuất các định hướng phát triển và nâng cấp tiềm năng cho hệ thống trong tương lai.

# CƠ SỞ LÝ THUYẾT

## Công nghệ Nền tảng (Core Technologies)

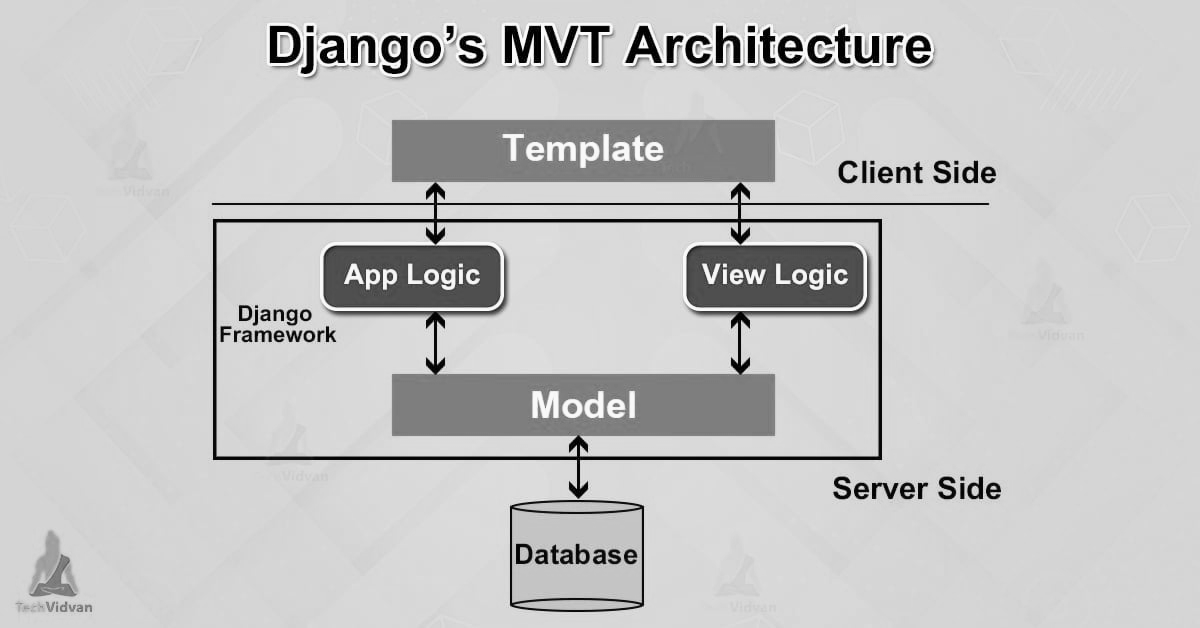
### Django và Django REST Framework (DRF)

#### Giới thiệu về Django Framework

Django là một web framework bậc cao, mã nguồn mở, được phát triển bằng ngôn ngữ Python. Khái niệm "framework" trong ngữ cảnh này chỉ một bộ khung sườn cấu trúc cùng với một tập hợp các thư viện và công cụ được xây dựng sẵn, nhằm mục đích đơn giản hóa và tăng tốc quá trình phát triển các ứng dụng web phức tạp. Bằng việc cung cấp các giải pháp đã được kiểm chứng cho những bài toán phổ biến, Django cho phép các nhà phát triển tập trung nguồn lực vào việc xây dựng logic nghiệp vụ đặc thù của sản phẩm, thay vì phải xây dựng lại từ đầu các thành phần nền tảng như quản lý cơ sở dữ liệu, xác thực người dùng, hay định tuyến các yêu cầu mạng.

#### Kiến trúc của Django Framework

Trong kiến trúc của hệ thống Landify, Django đóng vai trò là nền tảng cốt lõi của toàn bộ phía máy chủ (backend). Để điều phối mọi hoạt động một cách có tổ chức, Django tuân thủ một mẫu kiến trúc phần mềm gọi là MVT (Model-View-Template), một biến thể của mẫu MVC (Model-View-Controller) kinh điển. Kiến trúc này phân tách rõ ràng các thành phần của ứng dụng, và mỗi thành phần đảm nhiệm một vai trò cụ thể trong hệ thống:



Model: Là lớp định nghĩa cấu trúc dữ liệu, chịu trách nhiệm tương tác trực tiếp với cơ sở dữ liệu thông qua một thành phần mạnh mẽ gọi là Object-Relational Mapper (ORM). Trong Landify, các lớp như Listing, User, Property trong các file models.py chính là các Model. Chúng giúp trừu tượng hóa các câu lệnh SQL phức tạp, tăng tính an toàn và dễ bảo trì cho mã nguồn.

View: Là "bộ não" trung tâm, nơi tiếp nhận các yêu cầu HTTP từ client và thực thi toàn bộ logic nghiệp vụ. Đây là vai trò quan trọng nhất. Các View trong Landify xử lý mọi thứ, từ quy trình đăng tin nhiều bước, logic kiểm duyệt nội dung, xử lý các gói VIP, cho đến quản lý quyền hạn của người dùng. Nó sẽ tương tác với Model để lấy hoặc ghi dữ liệu trước khi quyết định phản hồi. Trong dự án này, các lớp ViewSet của DRF đóng vai trò là các View.

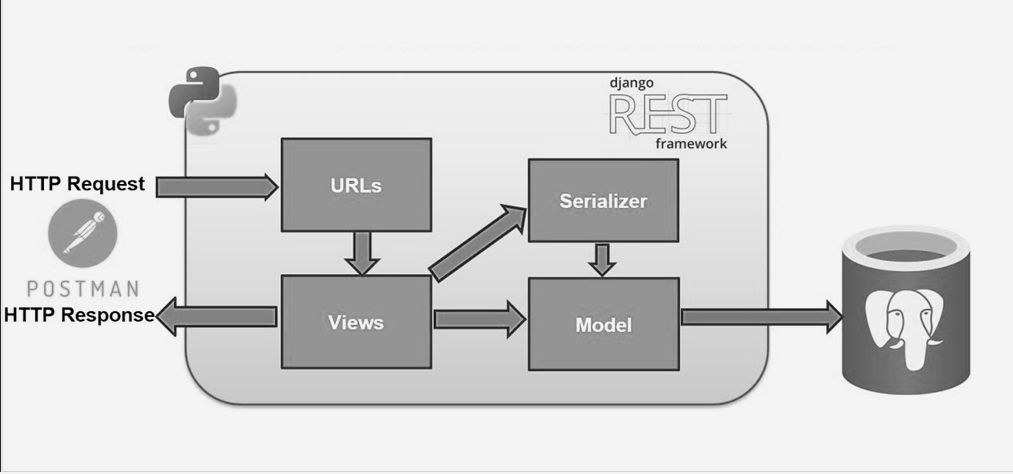
Template (hoặc Serializer trong API): Là lớp chịu trách nhiệm trình bày, định dạng dữ liệu để trả về phản hồi cho client. Trong một ứng dụng web truyền thống, đây là các file HTML. Tuy nhiên, trong một hệ thống API như Landify, vai trò này được đảm nhiệm bởi các Serializer của DRF, có nhiệm vụ chuyển đổi các đối tượng dữ liệu thành định dạng JSON để ứng dụng Flutter có thể hiểu và hiển thị cho người dùng.

#### Giới thiệu về Django REST Framework (DRF)

Django REST Framework (DRF) là một thư viện mở rộng, một bộ công cụ mạnh mẽ và linh hoạt được xây dựng chuyên biệt để hoạt động trên nền tảng Django. Nếu Django cốt lõi có thể được sử dụng để xây dựng các trang web truyền thống (trả về giao diện HTML), thì DRF tập trung hoàn toàn vào việc xây dựng các Web API (Application Programming Interface). Một Web API có thể được xem như một "cổng giao tiếp" cho phép các ứng dụng và hệ thống khác nhau có thể "nói chuyện" và trao đổi dữ liệu với nhau một cách có cấu trúc, thường là thông qua định dạng JSON (JavaScript Object Notation).

#### Kiến trúc của Django REST Framework (DRF)

Để hiểu rõ cách DRF hoạt động, chúng ta cần nhìn vào luồng xử lý một yêu cầu (request) từ client và cách nó tạo ra một phản hồi (response). Kiến trúc của DRF được thiết kế theo từng lớp, mỗi lớp đảm nhận một nhiệm vụ riêng biệt, giúp mã nguồn trở nên rõ ràng và dễ bảo trì. Luồng xử lý cơ bản diễn ra như sau:



Request (Yêu cầu) từ Client: Một ứng dụng client (ví dụ: ứng dụng Flutter) gửi một yêu cầu HTTP đến một URL cụ thể (ví dụ: GET /api/listings/).

URL Routing (Định tuyến URL): Django nhận yêu cầu và dựa vào cấu hình URL, nó sẽ chuyển yêu cầu này đến một View tương ứng do DRF quản lý. Các Router của DRF giúp tự động hóa quá trình tạo ra các URL này cho các tài nguyên một cách nhất quán.

View / ViewSet (Lớp xử lý): Đây là nơi logic chính được thực thi. View nhận yêu cầu, sử dụng các cơ chế tích hợp của DRF để thực hiện Authentication (Xác thực) và Permission (Kiểm tra quyền). Sau khi xác thực thành công, View sẽ tương tác với cơ sở dữ liệu thông qua Django ORM để lấy hoặc thay đổi dữ liệu.

Serializer (Lớp tuần tự hóa): Đây là "trái tim" của DRF.

* + Serialization (Đi): Khi View lấy được dữ liệu từ cơ sở dữ liệu (ví dụ: một đối tượng Listing), Serializer sẽ dịch đối tượng Python phức tạp này thành một định dạng dữ liệu đơn giản như JSON để gửi về cho client.
  + Deserialization (Đến): Khi client gửi dữ liệu lên (ví dụ: tạo một Listing mới qua phương thức POST), Serializer sẽ nhận dữ liệu JSON này, validate (xác thực) tính hợp lệ của nó (ví dụ: kiểm tra các trường bắt buộc, định dạng dữ liệu), và sau đó chuyển đổi nó trở lại thành một đối tượng Python để View có thể lưu vào cơ sở dữ liệu.

Response (Phản hồi): View sử dụng dữ liệu đã được tuần tự hóa từ Serializer để tạo ra một đối tượng Response của DRF, đóng gói nó trong một phản hồi HTTP (kèm theo status code, ví dụ: 200 OK hoặc 404 Not Found) và gửi trả lại cho client.

Trong kiến trúc Client-Server của Landify, DRF là thành phần không thể thiếu, đóng vai trò là "người phiên dịch" và "người điều phối" cho toàn bộ quá trình giao tiếp dữ liệu giữa ứng dụng Flutter và server Django.

#### Cách triển khai Django và DRF

Quá trình triển khai hệ thống backend với Django và Django REST Framework (DRF) là một quy trình có cấu trúc, bao gồm các bước từ thiết lập môi trường, xây dựng nền tảng, cho đến việc hiện thực hóa từng tính năng nghiệp vụ.

***Tổng quan các bước triển khai:***

1. Thiết lập Môi trường và Khởi tạo Dự án: Đây là bước nền tảng, tạo ra một không gian làm việc cô lập và cài đặt các thư viện cần thiết.
2. Cấu hình Dự án Cốt lõi: Định nghĩa các thiết lập cơ bản cho toàn bộ hệ thống, bao gồm kết nối cơ sở dữ liệu, đăng ký các thành phần cốt lõi và thiết lập các biến môi trường.
3. Xây dựng Module hóa theo "Apps": Áp dụng triết lý của Django để phân chia dự án thành các ứng dụng con, mỗi ứng dụng chịu trách nhiệm cho một phân hệ chức năng.
4. Thiết kế và Hiện thực hóa Lớp Dữ liệu (Models): Sử dụng Django ORM để định nghĩa cấu trúc cơ sở dữ liệu.
5. Xây dựng Lớp Giao tiếp API (Serializers và ViewSets): Sử dụng DRF để tạo ra các "cổng giao tiếp" cho phép frontend tương tác với dữ liệu.
6. Định tuyến và Hoàn thiện API: Kết nối các thành phần logic với các URL endpoint cụ thể.

***Chi tiết triển khai trong Landify:***

Bước 1: Thiết lập Môi trường và Khởi tạo Dự án

* + Tạo môi trường ảo (Virtual Environment): Để quản lý các gói thư viện một cách độc lập và tránh xung đột, một môi trường ảo Python đã được tạo bằng lệnh python -m venv .venv. Mọi thư viện sau đó đều được cài đặt vào môi trường này.
  + Cài đặt các thư viện lõi: Các thư viện nền tảng được cài đặt thông qua pip, bao gồm:

pip install django djangorestframework psycopg2-binary python-dotenv

Trong đó, django là framework chính, djangorestframework (DRF) dùng để xây dựng API, psycopg2-binary là driver để kết nối với PostgreSQL, và python-dotenv dùng để quản lý các biến môi trường.

* + Khởi tạo dự án và app đầu tiên: Dự án Django được khởi tạo bằng lệnh django-admin startproject landifyapis. Sau đó, các app chức năng đầu tiên như users và common được tạo bằng lệnh python manage.py startapp <app\_name>.

Bước 2: Cấu hình Dự án Cốt lõi

* + Quản lý biến môi trường: Một file .env được tạo ở thư mục gốc để lưu trữ các thông tin nhạy cảm như SECRET\_KEY, thông tin đăng nhập cơ sở dữ liệu (DB\_NAME, DB\_USER...), và các API key của dịch vụ bên thứ ba. Trong file settings.py, thư viện dotenv được sử dụng để đọc các biến này vào môi trường của ứng dụng.
  + Cấu hình Cơ sở dữ liệu: Trong settings.py, mục DATABASES được cấu hình để kết nối đến PostgreSQL, sử dụng các biến môi trường đã được định nghĩa.
  + Đăng ký Apps và Middleware: Các app mới tạo (apps.common, apps.users) và các thư viện của bên thứ ba (rest\_framework, corsheaders) được thêm vào danh sách INSTALLED\_APPS. Tương tự, các middleware cần thiết như CorsMiddleware (để cho phép request từ frontend) được thêm vào MIDDLEWARE.

Bước 3: Xây dựng Lớp Dữ liệu với Django Models

* + Định nghĩa Models: Trong file models.py của từng app, các lớp model được tạo để ánh xạ tới các bảng trong cơ sở dữ liệu. Ví dụ, trong apps/users/models.py, lớp User được tạo bằng cách kế thừa từ AbstractUser của Django để có thể thêm các trường tùy chỉnh như role và is\_identity\_verified. Mối quan hệ Một-Một được thiết lập với UserProfile để lưu trữ thông tin mở rộng.
  + Tạo và Áp dụng Migrations: Sau khi định nghĩa hoặc thay đổi models, hai lệnh quan trọng được thực thi:
    1. python manage.py makemigrations: Django tự động phân tích các thay đổi trong models.py và tạo ra các file "di trú" (migration files), chứa các chỉ dẫn để thay đổi cấu trúc cơ sở dữ liệu.
    2. python manage.py migrate: Lệnh này sẽ đọc các file migration và thực thi các câu lệnh SQL tương ứng để cập nhật cơ sở dữ liệu PostgreSQL.

Bước 4: Xây dựng Lớp Giao tiếp API với DRF

* + Tạo Serializers: Trong file serializers.py của mỗi app, các lớp Serializer được tạo để định nghĩa cách dữ liệu được biểu diễn. Ví dụ, UserSerializer trong apps/users/serializers.py kế thừa từ serializers.ModelSerializer và chỉ định các trường của model User sẽ được chuyển đổi thành JSON. Các kỹ thuật nâng cao như lồng serializer (UserProfileSimpleSerializer trong UserSerializer) cũng được áp dụng để trả về dữ liệu có cấu trúc.
  + Tạo ViewSets: Trong file views.py, các ViewSet được tạo để xử lý logic cho các API endpoint. Ví dụ, UserViewSet kế thừa từ viewsets.ModelViewSet, cung cấp sẵn các hành động CRUD (Create, Retrieve, Update, Delete). Các action tùy chỉnh như @action(methods=["get"], detail=False, url\_path="current-user") được thêm vào để tạo ra các endpoint đặc thù không theo chuẩn CRUD.
  + Tách biệt logic nghiệp vụ: Để giữ cho ViewSet gọn gàng, các logic phức tạp được chuyển vào file services.py. Ví dụ, UserViewSet sẽ gọi hàm accounts\_services.toggle\_user\_follow thay vì tự mình xử lý logic theo dõi người dùng.

Bước 5: Định tuyến URL và Hoàn thiện

* + Đăng ký Routers: Trong file urls.py của app, một DefaultRouter của DRF được khởi tạo. Lệnh router.register("users", views.UserViewSet) được sử dụng để đăng ký UserViewSet với router.
  + Kết nối URL toàn cục: Trong file landifyapis/urls.py chính của dự án, các URL pattern từ các app con được include vào. Điều này tạo ra một hệ thống định tuyến tập trung, ví dụ: mọi request đến /api/users/ sẽ được chuyển đến router của app users để xử lý.
  + Tích hợp Tài liệu API: Thư viện drf-spectacular được cấu hình trong settings.py và các URL của nó được thêm vào landifyapis/urls.py. Điều này cho phép hệ thống tự động quét toàn bộ các ViewSet và Serializer để sinh ra một giao diện tài liệu Swagger UI tương tác, giúp cho việc phát triển và kiểm thử API trở nên dễ dàng hơn.

Quá trình triển khai này, đi từ việc thiết lập môi trường, cấu hình, xây dựng từng lớp (Model, Serializer, View) cho đến khi hoàn thiện API, đảm bảo rằng hệ thống backend của Landify được xây dựng trên một nền tảng vững chắc, có cấu trúc rõ ràng và tuân thủ các quy ước tốt nhất của cả Django và Django REST Framework.

#### Lý do lựa chọn Django và Django REST Framework

Việc lựa chọn Django và Django REST Framework (DRF) làm nền tảng công nghệ cho hệ thống backend của Landify không phải là một quyết định ngẫu nhiên, mà là kết quả của một quá trình phân tích chiến lược, dựa trên các yêu cầu nghiệp vụ phức tạp và mục tiêu phát triển dài hạn của dự án. Sự kết hợp này đã cung cấp một bộ khung vững chắc, giải quyết hiệu quả các thách thức cốt lõi về tốc độ phát triển, khả năng tích hợp, bảo mật và khả năng mở rộng.

Ưu điểm

Tăng tốc phát triển hệ sinh thái phức tạp với triết lý "Batteries-included"

* + Thách thức: Landify cần xây dựng một hệ sinh thái đa chức năng (quản lý người dùng, tin đăng, tương tác xã hội, kiểm duyệt) trong thời gian giới hạn. Việc lựa chọn một framework có khả năng tăng tốc phát triển là yếu tố tiên quyết.
  + Giải pháp của Django: Django, với triết lý "Batteries-included" (có sẵn pin), cung cấp một bộ công cụ cực kỳ toàn diện. Thay vì phải xây dựng lại từ đầu, dự án đã tận dụng các thành phần có sẵn như:
    1. Hệ thống xác thực người dùng: Tích hợp sẵn, giúp xử lý đăng ký, đăng nhập và quản lý phiên một cách an toàn.
    2. ORM (Object-Relational Mapper): Cho phép tương tác với cơ sở dữ liệu PostgreSQL bằng mã Python, trừu tượng hóa các câu lệnh SQL phức tạp.
    3. Admin Panel (Trang quản trị): Đây là một ưu điểm vượt trội. Django tự động sinh ra một trang quản trị mạnh mẽ, trở thành công cụ vận hành nội bộ hiệu quả cho Landify, giúp đội ngũ quản trị viên kiểm duyệt tin đăng và quản lý dữ liệu mà không cần xây dựng một dashboard riêng biệt. Điều này cho phép nguồn lực dự án được tập trung hoàn toàn vào việc phát triển các tính năng nghiệp vụ cốt lõi.

Xây dựng API chuẩn hóa, linh hoạt và dễ tích hợp với DRF

* + Thách thức: Với kiến trúc Client-Server, việc xây dựng một hệ thống API (Application Programming Interface) chuẩn hóa, ổn định và có tài liệu rõ ràng là yêu cầu bắt buộc. API hoạt động như một "hợp đồng" (contract) vững chắc, cho phép hai đội ngũ backend và frontend phát triển song song.
  + Giải pháp của DRF: Django REST Framework (DRF) là tiêu chuẩn công nghiệp khi xây dựng API với Django. Nó mở rộng kiến trúc MVT (Model-View-Template) của Django để phục vụ cho việc giao tiếp dữ liệu:
    1. Kiến trúc luồng xử lý của DRF: DRF xử lý một yêu cầu từ client (Flutter) qua các lớp: URL Routing -> ViewSet (xử lý logic, tương đương View trong MVT) -> Serializer (chuyển đổi dữ liệu, tương đương Template trong MVT) -> Response.
    2. Tự động hóa và chuẩn hóa: Việc sử dụng các thành phần như ViewSet và Router đã tự động hóa quá trình tạo ra các URL endpoint nhất quán, tuân thủ theo nguyên tắc kiến trúc REST.
    3. Tài liệu API tự động: Khả năng tích hợp liền mạch với thư viện drf-spectacular để tự động sinh ra tài liệu Swagger UI là một lợi thế cực kỳ lớn. Giao diện tài liệu tương tác này đã cung cấp một nguồn tham chiếu chính xác, luôn được cập nhật cho đội ngũ frontend, giúp giảm thiểu lỗi giao tiếp và đẩy nhanh tiến độ tích hợp.

Đảm bảo tính bảo mật và toàn vẹn dữ liệu

* + **Thách thức:** Là một nền tảng có các hoạt động giao dịch và xử lý thông tin cá nhân, Landify đặt ra yêu cầu rất cao về bảo mật và sự toàn vẹn của dữ liệu.
  + **Giải pháp của Django:**
    1. Bảo mật tích hợp sẵn: Django cung cấp các cơ chế phòng chống hiệu quả trước các lỗ hổng phổ biến như SQL Injection, Cross-Site Scripting (XSS), và Cross-Site Request Forgery (CSRF).
    2. Toàn vẹn dữ liệu với Giao dịch (Transactions): Django ORM đóng vai trò quan trọng trong việc đảm bảo tính nguyên tử (atomicity). Thông qua transaction.atomic, các nghiệp vụ phức tạp (ví dụ: tạo một tin đăng mới gồm nhiều bản ghi liên quan) được đảm bảo sẽ thành công hoàn toàn hoặc thất bại hoàn toàn. Điều này ngăn chặn tuyệt đối tình trạng dữ liệu bị "mồ côi" hay không nhất quán.

Nền tảng cho khả năng mở rộng và bảo trì trong tương lai

* + **Thách thức:** Landify được định hướng là một hệ thống có tiềm năng phát triển và mở rộng trong tương lai.
  + **Giải pháp của Django:** Kiến trúc module hóa theo **"apps"** là nền tảng cho khả năng này. Việc phân chia rõ ràng các phân hệ chức năng (users, listings, interactions...) thành các module độc lập giúp việc bảo trì, gỡ lỗi và nâng cấp trở nên dễ dàng. Khi cần thêm một tính năng mới, nhóm phát triển có thể tạo ra một app riêng biệt mà ít gây ảnh hưởng đến các phần còn lại của hệ thống.

Nhược điểm

**Tính nguyên khối (Monolithic):** Django có thể trở nên cồng kềnh đối với các dự án siêu nhỏ. Tuy nhiên, với sự phức tạp và quy mô của Landify, cấu trúc này lại là một lợi thế, cung cấp sự nhất quán và đầy đủ tính năng.

**Đường cong học tập (Learning Curve):** Mặc dù cú pháp Python dễ tiếp cận, việc làm chủ toàn bộ hệ sinh thái và các khái niệm trừu tượng của Django và DRF đòi hỏi một khoảng thời gian đầu tư. Đây là một sự đánh đổi cần thiết để có được một hệ thống mạnh mẽ và có cấu trúc.

Như vậy ta có thể thấy được rằng: Sự kết hợp giữa Django và Django REST Framework đã cung cấp cho Landify một bộ khung backend vừa mạnh mẽ, an toàn, vừa linh hoạt. Nó không chỉ đáp ứng đầy đủ các yêu cầu chức năng phức tạp của dự án ở thời điểm hiện tại mà còn đặt một nền móng kiến trúc vững chắc, sẵn sàng cho sự phát triển và mở rộng của hệ thống trong tương lai.

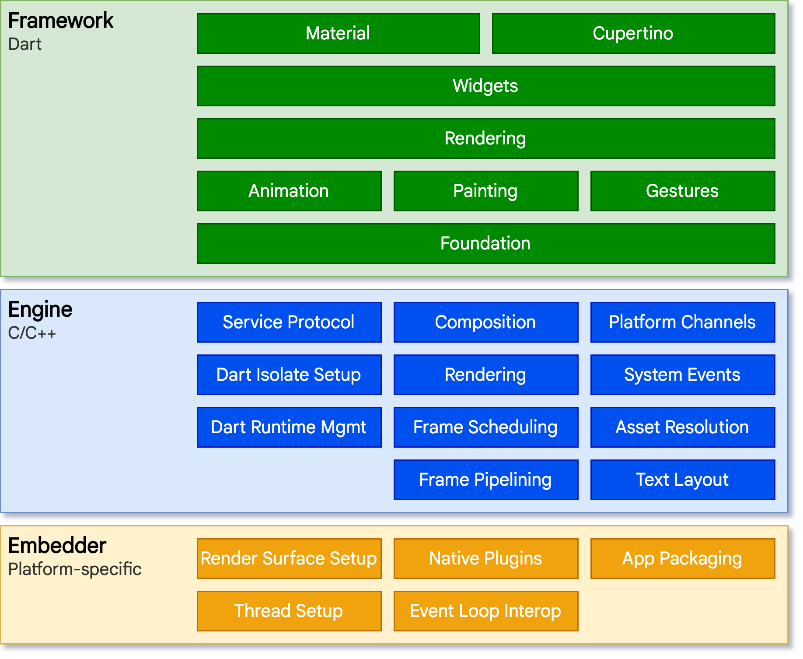
### Flutter

#### Giới thiệu về Flutter Framework

Flutter là một bộ công cụ phát triển giao diện người dùng (UI toolkit) mã nguồn mở, được tạo ra và phát triển bởi Google. Về bản chất, Flutter không phải là một ngôn ngữ lập trình, mà là một framework toàn diện, cung cấp cho các nhà phát triển một tập hợp các công cụ, thư viện và widget (thành phần giao diện) để xây dựng các ứng dụng có giao diện đẹp mắt và hiệu năng cao. Điểm đặc biệt nhất của Flutter là khả năng biên dịch ứng dụng từ một cơ sở mã nguồn duy nhất (single codebase) ra các tệp thực thi gốc (native) cho nhiều nền tảng khác nhau, bao gồm di động (iOS, Android), web, và máy tính để bàn (Windows, macOS, Linux).

#### Kiến trúc của Flutter

Sức mạnh của Flutter đến từ kiến trúc phân lớp (layered architecture), cho phép các nhà phát triển kiểm soát mọi pixel trên màn hình và đạt được hiệu năng cao. Kiến trúc này bao gồm ba lớp chính:



* Lớp Framework (viết bằng Dart): Đây là lớp mà các nhà phát triển ứng dụng tương tác trực tiếp nhiều nhất. Nó được viết hoàn toàn bằng ngôn ngữ Dart và chứa một bộ công cụ phong phú để xây dựng giao diện người dùng. Lớp này bao gồm nhiều tầng nhỏ hơn:
  + Foundation: Cung cấp các khối xây dựng cơ bản và các tiện ích cấp thấp (như ChangeNotifier).
  + Animation, Painting, Gestures: Các lớp nền tảng để xử lý hoạt ảnh, vẽ giao diện lên canvas, và nhận diện cử chỉ tương tác của người dùng.
  + Rendering: Chịu trách nhiệm xây dựng cây đối tượng kết xuất (render tree), thực hiện việc tính toán bố cục và "tô vẽ" giao diện.
  + Widgets: Đây là lớp trừu tượng hóa của lớp Rendering, chứa một thư viện khổng lồ các thành phần giao diện. Toàn bộ giao diện được xây dựng dựa trên triết lý cốt lõi "Mọi thứ đều là Widget".
  + Material và Cupertino: Nằm ở tầng cao nhất, cung cấp các bộ widget được thiết kế sẵn theo ngôn ngữ thiết kế của Google (Material) và Apple (Cupertino).
* Lớp Engine (viết bằng C++/Dart): Đây là "trái tim" của Flutter, cung cấp các triển khai cấp thấp cho toàn bộ các tính năng cốt lõi. Engine được viết chủ yếu bằng C++ để đảm bảo hiệu năng cao và sử dụng thư viện đồ họa 2D Skia để vẽ mọi pixel lên màn hình. Các thành phần chính của Engine bao gồm:
  + Dart Runtime & Isolate Setup: Cung cấp môi trường để thực thi mã Dart.
  + Rendering, Text Layout, Composition: Xử lý các tác vụ đồ họa cấp thấp, từ bố cục văn bản cho đến tổng hợp các lớp giao diện để gửi đến Skia.
  + Frame Scheduling & Pipelining: Quản lý vòng lặp sự kiện và lịch trình vẽ các khung hình để đạt được hoạt ảnh mượt mà.
  + Platform Channels: Đây là "cây cầu" cho phép lớp Framework (Dart) giao tiếp với lớp Embedder (Platform-specific).
  + Service Protocol, Asset Resolution, System Events: Các thành phần khác để quản lý tài nguyên, sự kiện hệ thống và gỡ lỗi.

**Lớp Embedder (Platform-specific):**

Đây là lớp nhúng dành riêng cho từng nền tảng, được viết bằng ngôn ngữ gốc (Kotlin/Java cho Android, Swift/Objective-C cho iOS). Nó đóng vai trò là cầu nối để tích hợp Flutter vào hệ điều hành chủ. Nhiệm vụ của nó bao gồm:

* + Render Surface Setup: Thiết lập một bề mặt để Flutter có thể vẽ lên.
  + Thread Setup: Quản lý các luồng (threads), bao gồm luồng giao diện chính của nền tảng (platform UI thread) và luồng giao diện của Flutter (Flutter UI thread).
  + Event Loop Interop: Tích hợp vòng lặp sự kiện của Flutter với vòng lặp sự kiện của hệ điều hành.
  + Native Plugins: Quản lý việc giao tiếp với các plugin gốc thông qua Platform Channels.
  + App Packaging: Đóng gói ứng dụng thành một tệp thực thi có thể chạy được trên hệ điều hành tương ứng.

#### *Vai trò của Flutter*

Trong dự án Landify, Flutter đóng vai trò là công nghệ nền tảng để xây dựng toàn bộ phía Client của hệ thống – tức là ứng dụng di động mà người dùng cuối sẽ trực tiếp tương tác. Vai trò của nó bao gồm:

Xây dựng Giao diện Người dùng (UI): Flutter chịu trách nhiệm render (kết xuất) toàn bộ các thành phần giao diện mà người dùng nhìn thấy và tương tác, từ các nút bấm, ô nhập liệu, danh sách tin đăng cho đến các màn hình phức tạp như bản đồ hay giao diện chat.

Quản lý Trạng thái Ứng dụng: Nó cung cấp cơ chế để quản lý trạng thái của ứng dụng (ví dụ: người dùng đã đăng nhập hay chưa, danh sách tin đăng đã được tải về hay chưa) và tự động cập nhật giao diện khi trạng thái này thay đổi.

Xử lý Tương tác Người dùng: Flutter bắt và xử lý mọi tương tác của người dùng như nhấn, cuộn, kéo, sau đó chuyển các tương tác này thành các sự kiện logic để xử lý ở tầng nghiệp vụ.

Giao tiếp với Backend: Thông qua các thư viện mạng, ứng dụng Flutter tạo và gửi các yêu cầu HTTP đến hệ thống API của Django để lấy dữ liệu hoặc thực hiện các hành động, sau đó nhận và phân tích dữ liệu JSON trả về.

#### Cách triển khai Flutter

Việc triển khai Flutter cho ứng dụng di động Landify được thực hiện theo một quy trình có hệ thống, tuân thủ các nguyên tắc kiến trúc phần mềm hiện đại nhằm đảm bảo mã nguồn có tổ chức, dễ quản lý và có khả năng mở rộng. Quá trình này không chỉ là việc xây dựng giao diện, mà còn là việc thiết kế một luồng dữ liệu và quản lý trạng thái chặt chẽ.

Tổng quan các bước triển khai:

1. Thiết lập Môi trường và Cấu trúc Dự án: Chuẩn bị các công cụ cần thiết và định hình cấu trúc thư mục nền tảng cho toàn bộ dự án.
2. Xây dựng Tầng Giao tiếp (API Client & Repositories): Tạo ra một lớp trừu tượng để ứng dụng có thể giao tiếp với backend API một cách nhất quán.
3. Thiết kế Tầng Dữ liệu (Models & ViewModels): Áp dụng mô hình MVVM để phân tách dữ liệu thô khỏi dữ liệu đã được định dạng cho giao diện.
4. Hiện thực hóa Tầng Logic (BLoCs): Sử dụng kiến trúc BLoC để quản lý trạng thái và xử lý logic nghiệp vụ cho từng tính năng.
5. Xây dựng Tầng Giao diện (Screens & Widgets): Kết hợp các widget để tạo ra các màn hình hoàn chỉnh, chỉ chịu trách nhiệm hiển thị dữ liệu và gửi sự kiện.
6. Thiết lập Điều hướng (Routing): Định nghĩa và quản lý luồng di chuyển giữa các màn hình trong ứng dụng.

***Chi tiết triển khai trong Landify:***

Bước 1: Thiết lập Môi trường và Cấu trúc Dự án

* + Cài đặt Flutter SDK và các công cụ: Môi trường phát triển được thiết lập với Flutter SDK, cùng với các plugin cần thiết cho IDE (Visual Studio Code hoặc Android Studio) và các công cụ dòng lệnh như FlutterFire CLI để tích hợp Firebase.
  + Tổ chức cấu trúc thư mục: Dự án được cấu trúc theo một quy ước rõ ràng. Thư mục lib/ được chia thành hai phần chính:
    1. core/: Chứa các thành phần dùng chung cho toàn bộ ứng dụng như api/ (quản lý kết nối API), auth/ (xử lý xác thực), models/, view\_models/, widgets/ (các widget tái sử dụng), và theme/ (định nghĩa giao diện chung).
    2. features/: Chứa các module chức năng độc lập. Mỗi feature (ví dụ: listing\_create, chat\_and\_call) lại được chia nhỏ thành các tầng data (repositories, models), logic (blocs), và presentation (screens, widgets).

Bước 2: Xây dựng Tầng Giao tiếp (API Client & Repositories)

* + ApiClient: Một lớp ApiClient (core/api/api\_client.dart) được tạo ra bằng thư viện Dio. Lớp này đóng vai trò là một singleton, quản lý các cấu hình chung cho việc gọi API như baseUrl, timeout. Quan trọng nhất, nó sử dụng Interceptors để tự động đính kèm Firebase ID Token vào header của mỗi request yêu cầu xác thực, giúp mã nguồn ở tầng Repository không cần lặp lại logic này.
  + Repositories: Đối với mỗi nhóm nghiệp vụ, một lớp Repository tương ứng được tạo ra (ví dụ: ListingDetailRepository, WishlistRepository). Vai trò của Repository là trừu tượng hóa việc lấy dữ liệu. Nó sẽ gọi đến ApiClient, xử lý các lỗi mạng cơ bản, và parse dữ liệu JSON trả về thành các Data Models.

Bước 3: Thiết kế Tầng Dữ liệu (Models & ViewModels)

* + Data Models: Trong core/models/, các lớp như ListingPreviewModel hay UserModel được tạo ra. Các lớp này có cấu trúc ánh xạ 1-1 với JSON trả về từ API và chỉ chứa một factory constructor fromJson để thực hiện việc parse dữ liệu.
  + ViewModels: Trong thư mục presentation/view\_models của mỗi feature, các lớp ViewModel được tạo ra. Ví dụ, ListingPreviewViewModel nhận vào một ListingPreviewModel và "chế biến" dữ liệu, chẳng hạn như chuyển đổi createdDate thành chuỗi "Đăng hôm qua" hoặc định dạng priceValue thành "1.5 tỷ".

Bước 4: Hiện thực hóa Tầng Logic với BLoC (Business Logic Component)

* + Đối với mỗi tính năng hoặc màn hình có trạng thái phức tạp, một BLoC tương ứng được tạo ra (ví dụ: HomepageBloc, ChatBloc, VerificationBloc).
  + Events và States: Thư viện freezed được sử dụng để định nghĩa các lớp Event (đại diện cho hành động của người dùng) và State (đại diện cho trạng thái của giao diện).
  + Xử lý Logic: Bên trong mỗi BLoC, các handler sẽ lắng nghe Event, gọi đến Repository để lấy Data Models, sau đó tạo ra các ViewModels và cuối cùng là emit ra một State mới chứa các ViewModel đó. Ví dụ, ListingDetailBloc khi nhận ListingDetailFetched event sẽ gọi ListingDetailRepository, nhận về ListingDetailModel, chuyển nó thành ListingDetailViewModel, và phát ra ListingDetailState chứa ViewModel này.

Bước 5: Xây dựng Tầng Giao diện với Widgets

* + Giao diện của Landify được xây dựng hoàn toàn bằng cách kết hợp các Widget. Các màn hình lớn (Screens) được đặt trong thư mục presentation/screens, trong khi các thành phần nhỏ hơn, có thể tái sử dụng (Widgets) được đặt trong presentation/widgets.
  + Các widget giao diện được thiết kế để "ngu ngốc" (dumb components). Chúng chỉ nhận dữ liệu từ State của BLoC (thông qua BlocBuilder) và gửi các Event đến BLoC khi có tương tác. Ví dụ, widget PropertyCard nhận vào một ListingPreviewViewModel để hiển thị, và khi người dùng nhấn nút yêu thích, nó sẽ gửi một ToggleWishlistEvent đến ToggleWishlistBloc.

Bước 6: Thiết lập Điều hướng (Routing)

* + Luồng di chuyển giữa các màn hình được quản lý tập trung trong file app.dart.
  + Các route tĩnh (không có tham số) được định nghĩa trong thuộc tính routes của MaterialApp.
  + Các route động (cần truyền tham số, ví dụ như ID của tin đăng) được xử lý trong hàm onGenerateRoute. Hàm này sẽ phân tích RouteSettings để trích xuất tham số và khởi tạo màn hình tương ứng với dữ liệu cần thiết, ví dụ: ListingDetailPage(listingId: listingId).

Bằng cách tuân thủ nghiêm ngặt kiến trúc nhiều tầng và các nguyên tắc thiết kế hiện đại này, ứng dụng Flutter của Landify đã đạt được sự phân tách rõ ràng giữa các thành phần, giúp mã nguồn trở nên dễ đọc, dễ kiểm thử, và đặc biệt là dễ dàng bảo trì và mở rộng trong tương lai.

#### Lý do lựa chọn Flutter

Ưu điểm

Quyết định sử dụng Flutter làm framework chính để phát triển ứng dụng di động Landify được đưa ra sau khi cân nhắc kỹ lưỡng các yêu cầu về sản phẩm, mục tiêu kinh doanh và nguồn lực phát triển. Flutter không chỉ là một lựa chọn công nghệ, mà còn là một quyết định chiến lược nhằm giải quyết các thách thức cốt lõi của dự án.

Tối ưu hóa Nguồn lực và Tăng tốc độ ra mắt sản phẩm

* + Thách thức: Một trong những rào cản lớn nhất của bất kỳ dự án khởi nghiệp nào là tối ưu hóa nguồn lực và thời gian để đưa sản phẩm ra thị trường (go-to-market) nhanh nhất có thể. Việc phát triển hai ứng dụng riêng biệt cho iOS và Android sẽ đòi hỏi gấp đôi thời gian, chi phí và nhân lực.
  + Giải pháp của Flutter: Với khả năng phát triển đa nền tảng từ một cơ sở mã nguồn duy nhất ("viết một lần, chạy mọi nơi"), Flutter đã giải quyết triệt để bài toán này. Thay vì phải duy trì hai đội ngũ, dự án tập trung vào một codebase duy nhất bằng ngôn ngữ Dart. Hơn nữa, tính năng đột phá "Stateful Hot Reload" cho phép thấy kết quả thay đổi mã nguồn gần như ngay lập tức, giúp rút ngắn đáng kể chu trình phát triển-kiểm thử. Điều này không chỉ giảm chi phí mà còn đảm bảo các tính năng mới và bản vá lỗi được cập nhật đồng bộ trên cả hai hệ điều hành, tăng tốc độ ra mắt sản phẩm.

Đảm bảo Trải nghiệm Người dùng Mượt mà và Đồng nhất

* + Yêu cầu: Đối với một ứng dụng hướng đến người dùng cuối như Landify, trải nghiệm người dùng (UX) là yếu tố sống còn. Các thao tác cuộn danh sách, chuyển cảnh, và hoạt ảnh phải mượt mà để tạo cảm giác chuyên nghiệp và đáng tin cậy.
  + Giải pháp của Flutter: Flutter được lựa chọn vì khả năng cung cấp hiệu năng tương đương ứng dụng gốc. Bằng cách sử dụng engine đồ họa Skia để tự vẽ mọi pixel trên màn hình và biên dịch mã Dart ra mã máy gốc (Ahead-Of-Time), ứng dụng Landify có thể đạt được tốc độ 60 khung hình/giây (fps) một cách ổn định. Kiến trúc dựa trên widget của Flutter cũng đảm bảo giao diện của Landify trông và hoạt động hoàn toàn giống nhau trên cả thiết bị iOS và Android, tạo ra một trải nghiệm thương hiệu đồng nhất và mạnh mẽ.

Kiến trúc Phù hợp cho một Ứng dụng Phức tạp và Dài hạn

* + Thách thức: Landify là một hệ thống phức tạp với nhiều màn hình, luồng nghiệp vụ và trạng thái dữ liệu thay đổi liên tục. Một kiến trúc phần mềm rõ ràng là yêu cầu bắt buộc để quản lý sự phức tạp này và đảm bảo dự án có thể được bảo trì, mở rộng trong dài hạn.
  + Giải pháp của Flutter: Hệ sinh thái của Flutter, đặc biệt là sự hỗ trợ mạnh mẽ cho các mẫu kiến trúc hiện đại như BLoC (Business Logic Component), đã cung cấp một giải pháp hoàn hảo. Việc áp dụng kiến trúc BLoC kết hợp với MVVM (Model-View-ViewModel) đã giúp phân tách hoàn toàn logic nghiệp vụ khỏi giao diện người dùng. Điều này làm cho mã nguồn của từng tính năng trở nên độc lập, dễ đọc, dễ kiểm thử, và giúp cho việc thêm tính năng mới hoặc thay đổi yêu cầu trong tương lai trở nên dễ dàng hơn rất nhiều.

Cộng đồng Phát triển Mạnh mẽ và Hệ sinh thái Phong phú

* + Yêu cầu: Tốc độ phát triển của một dự án phụ thuộc rất nhiều vào sự hỗ trợ từ cộng đồng và hệ sinh thái các thư viện có sẵn.
  + Giải pháp của Flutter: Được hậu thuẫn bởi Google và có một trong những cộng đồng mã nguồn mở phát triển nhanh nhất, Flutter mang lại hai lợi ích lớn. Thứ nhất, việc tìm kiếm giải pháp cho các vấn đề kỹ thuật trở nên dễ dàng. Thứ hai, dự án có thể tận dụng một kho thư viện phong phú trên pub.dev để tích hợp nhanh các tính năng phức tạp, từ việc gọi API (dio), quản lý trạng thái (flutter\_bloc), đến tích hợp các SDK của bên thứ ba như Firebase, Agora.

Khuyết điểm

Dù có nhiều ưu điểm, nhóm phát triển cũng đã cân nhắc các khuyết điểm của Flutter:

Kích thước Ứng dụng: Do phải đóng gói cả engine Skia, kích thước tệp cài đặt của ứng dụng Flutter thường lớn hơn một chút so với ứng dụng gốc. Đây là một sự đánh đổi được chấp nhận, vì ưu tiên hàng đầu của dự án là tốc độ phát triển và trải nghiệm người dùng, và dung lượng lưu trữ trên các thiết bị hiện đại không còn là rào cản lớn.

Hệ sinh thái Thư viện: Mặc dù phát triển nhanh, hệ sinh thái thư viện vẫn chưa thể phong phú bằng các nền tảng gốc lâu đời. Tuy nhiên, đối với các tính năng cốt lõi của Landify, hệ sinh thái hiện tại đã cung cấp đầy đủ các thư viện chất lượng cao. Các tính năng rất đặc thù có thể được giải quyết bằng Kênh Nền tảng (Platform Channels) khi cần thiết.

Tóm lại, việc lựa chọn Flutter không chỉ là một quyết định về mặt công nghệ, mà còn là một lựa chọn chiến lược. Nó giúp Landify đạt được sự cân bằng tối ưu giữa tốc độ phát triển, chất lượng sản phẩm, hiệu năng và khả năng bảo trì, tạo ra một nền tảng vững chắc để xây dựng một ứng dụng di động bất động sản cạnh tranh và thành công.

### PostgreSQL

#### Giới thiệu về PostgreSQL

PostgreSQL, thường được gọi là "Postgres", là một hệ quản trị cơ sở dữ liệu quan hệ đối tượng (Object-Relational Database Management System - ORDBMS) mã nguồn mở mạnh mẽ và tiên tiến. "Cơ sở dữ liệu quan hệ" có nghĩa là dữ liệu được tổ chức một cách logic thành các bảng (tables), trong đó mỗi bảng bao gồm các hàng (rows) và các cột (columns), và các mối quan-hệ giữa các bảng này được định nghĩa một cách chặt chẽ. PostgreSQL nổi tiếng trong cộng đồng phát triển nhờ sự tuân thủ nghiêm ngặt các tiêu chuẩn SQL, độ tin cậy cao, và một bộ tính năng cực kỳ phong phú đã được phát triển và hoàn thiện trong hơn 30 năm.

#### Kiến trúc của PostgreSQL

Kiến trúc của PostgreSQL vận hành theo mô hình client-server, bao gồm bốn thành phần chính được phân chia một cách logic: Client, Server Processes (Các tiến trình máy chủ), System Memory (Bộ nhớ hệ thống), và Physical Files (Lưu trữ vật lý). Toàn bộ sự kết hợp của các tiến trình và bộ nhớ tạo nên một thực thể gọi là PostgreSQL Instance.

Ảnh có chứa văn bản, ảnh chụp màn hình, biểu đồ, hàng

Nội dung do AI tạo ra có thể không chính xác.

Tương tác Client-Server và các Tiến trình Xử lý

Luồng tương tác bắt đầu từ phía Client và được điều phối bởi các tiến trình trên Server:

* + Client: Bao gồm Ứng dụng Client (ví dụ: ứng dụng Django) và Thư viện Giao tiếp Client (ví dụ: psycopg2). Ứng dụng gửi các yêu cầu CSDL (DB Requests) thông qua thư viện này.
  + Tiến trình Chủ (Postmaster Daemon Process): Khi Client khởi tạo một kết nối, yêu cầu ban đầu (bao gồm xác thực và phân quyền) sẽ được gửi đến tiến trình chủ Postmaster trên cổng mặc định 5432. Đây là tiến trình "lắng nghe" và quản lý toàn bộ instance.
  + Tiến trình Backend (postgres backend process): Sau khi xác thực thành công, Postmaster sẽ không trực tiếp xử lý truy vấn. Thay vào đó, nó sẽ tạo ra (forks) một tiến trình backend postgres hoàn toàn mới để phục vụ riêng cho kết nối đó. Mọi tương tác sau này, như gửi các câu lệnh SQL và nhận kết quả, sẽ diễn ra trực tiếp giữa Thư viện Giao tiếp Client và tiến trình backend chuyên dụng này. Mô hình "mỗi kết nối một tiến trình" này đảm bảo sự cô lập và ổn định cho hệ thống.

Kiến trúc Bộ nhớ Hệ thống (System Memory)

Bộ nhớ của PostgreSQL được chia thành hai loại riêng biệt, phục vụ cho các mục đích khác nhau:

Bộ nhớ Chia sẻ (PostgreSQL Shared Memory): Đây là vùng bộ nhớ được chia sẻ chung cho tất cả các tiến trình trong instance (bao gồm cả Postmaster và các tiến trình nền). Nó chứa các thành phần cốt lõi để CSDL hoạt động hiệu quả:

* + Shared Buffers: Vùng đệm lớn nhất, lưu trữ các trang dữ liệu (data pages) được đọc từ đĩa. Đây là bộ nhớ đệm chính giúp giảm thiểu các thao tác I/O.
  + WAL Buffers (Write-Ahead Log Buffers): Vùng đệm chuyên dụng để tạm thời lưu trữ các bản ghi về sự thay đổi dữ liệu trước khi chúng được ghi xuống các file WAL.
  + CLOG Buffers (Commit Log Buffers): Lưu trữ trạng thái (đã commit, đã hủy) của các giao dịch, một thành phần thiết yếu cho cơ chế MVCC.
* Bộ nhớ Cục bộ cho từng Tiến trình (Per Backend Memory): Mỗi tiến trình backend postgres được cấp phát một vùng bộ nhớ riêng, không chia sẻ với các tiến trình khác. Vùng nhớ này được sử dụng để xử lý các truy vấn của riêng kết nối đó:
  + work\_mem: Dùng cho các hoạt động yêu cầu nhiều bộ nhớ như sắp xếp (ORDER BY) và băm (hash joins).
  + maintenance\_work\_mem: Dùng cho các tác vụ bảo trì như tạo chỉ mục (CREATE INDEX) hay dọn dẹp (VACUUM).
  + temp\_buffer: Dùng để lưu trữ các bảng tạm thời (temporary tables).

Các Tiến trình Nền (Utility Processes)

Để đảm bảo CSDL vận hành một cách trơn tru, tự động và bền vững, một loạt các tiến trình nền (hay còn gọi là tiến trình tiện ích) chạy liên tục ở hậu trường:

* + Checkpointer: Định kỳ ghi các trang dữ liệu đã bị thay đổi ("dirty pages") từ Shared Buffers xuống các file dữ liệu chính trên đĩa.
  + BGWriter (Background Writer): Hỗ trợ Checkpointer bằng cách ghi dần các buffer bẩn ra đĩa, giúp giảm tải công việc dồn dập tại các thời điểm checkpoint.
  + WAL Writer: Chuyên ghi các bản ghi từ WAL Buffers xuống các file WAL trên đĩa một cách hiệu quả.
  + Auto Vacuum: Tự động khởi chạy các tiến trình dọn dẹp để thu hồi không gian từ các phiên bản hàng đã lỗi thời do cơ chế MVCC tạo ra, đồng thời cập nhật các số liệu thống kê cho bộ tối ưu hóa truy vấn.
  + Archiver: Sao chép các file WAL đã hoàn thành sang một khu vực lưu trữ dự phòng, phục vụ cho việc sao lưu và phục hồi tại một thời điểm (Point-in-Time Recovery).

4. Lưu trữ Vật lý (Physical Files)

Toàn bộ dữ liệu, log và thông tin cấu hình của CSDL được lưu trữ bền vững trên đĩa dưới dạng các file có cấu trúc:

* + Data Files: Nơi chứa dữ liệu thực tế của các bảng và chỉ mục.
  + WAL Files (Write-Ahead Log Files): Chuỗi các file ghi lại mọi thay đổi xảy ra với CSDL. Đây là thành phần cốt lõi để đảm bảo tính toàn vẹn dữ liệu (Durability) và khôi phục sau sự cố.
  + Archive Files: Bản sao của các WAL Files, được tạo bởi tiến trình Archiver.
  + Log Files: Các file log ghi lại hoạt động, lỗi và các thông tin chẩn đoán của server.

#### Vai trò của PostgreSQL

Trong kiến trúc hệ thống Landify, PostgreSQL đóng vai trò là kho lưu trữ dữ liệu trung tâm và bền vững. Nó là nơi lưu trữ toàn bộ thông tin có cấu trúc của ứng dụng, từ dữ liệu người dùng, thông tin chi tiết về bất động sản, nội dung các tin đăng, cho đến lịch sử các tương tác và giao dịch. Vai trò cụ thể của nó bao gồm:

Lưu trữ Dữ liệu có Cấu trúc: PostgreSQL chịu trách nhiệm lưu trữ tất cả các bản ghi của ứng dụng, chẳng hạn như thông tin trong các bảng User, Property, Listing, Appointment, và Message.

Đảm bảo Toàn vẹn Dữ liệu (Data Integrity): Thông qua các cơ chế như khóa chính (primary keys), khóa ngoại (foreign keys), các ràng buộc (constraints) và giao dịch (transactions), PostgreSQL đảm bảo rằng dữ liệu trong hệ thống luôn ở trạng thái nhất quán, chính xác và tuân thủ các quy tắc nghiệp vụ đã được định nghĩa.

Thực thi Truy vấn Dữ liệu: Nó thực thi các câu lệnh truy vấn (queries) được gửi đến từ ứng dụng Django (thông qua ORM) để đọc, ghi, sửa, và xóa dữ liệu. Hiệu năng của các truy vấn này ảnh hưởng trực tiếp đến tốc độ phản hồi của toàn bộ hệ thống API.

Hỗ trợ các Kiểu Dữ liệu Nâng cao: Không chỉ giới hạn ở các kiểu dữ liệu cơ bản như số hay chuỗi, PostgreSQL còn hỗ trợ các kiểu dữ liệu phức tạp hơn, tạo nền tảng cho các tính năng nâng cao của Landify.

#### Cách triển khai PostgreSQL

Việc tích hợp và sử dụng PostgreSQL làm cơ sở dữ liệu chính cho dự án Landify được thực hiện một cách liền mạch thông qua các công cụ và quy ước mà Django cung cấp. Quá trình này bao gồm các bước từ cấu hình kết nối, thiết kế mô hình dữ liệu, cho đến việc tương tác với cơ sở dữ liệu trong quá trình vận hành.

***Tổng quan các bước triển khai:***

1. Cài đặt và Cấu hình Môi trường: Chuẩn bị driver cần thiết và thiết lập các thông tin kết nối an toàn.
2. Định nghĩa Lược đồ Dữ liệu qua Django Models: Sử dụng Django ORM để mô hình hóa cơ sở dữ liệu bằng các lớp Python.
3. Quản lý Thay đổi Lược đồ với Migrations: Sử dụng hệ thống "di trú" (migrations) của Django để áp dụng và theo dõi các thay đổi về cấu trúc cơ sở dữ liệu một cách có kiểm soát.
4. Tương tác với Dữ liệu trong Ứng dụng: Thực hiện các thao tác CRUD (Create, Read, Update, Delete) và các truy vấn phức tạp thông qua Django ORM.
5. Tối ưu hóa Truy vấn: Áp dụng các kỹ thuật tối ưu hóa để đảm bảo hiệu năng của hệ thống khi tương tác với cơ sở dữ liệu.

***Chi tiết triển khai trong Landify:***

Bước 1: Cài đặt và Cấu hình Kết nối

* + Cài đặt Driver: Để ứng dụng Django có thể "nói chuyện" được với cơ sở dữ liệu PostgreSQL, một thư viện driver là psycopg2-binary đã được cài đặt vào môi trường ảo của dự án thông qua lệnh pip install psycopg2-binary.
  + Cấu hình Kết nối An toàn: Thay vì ghi trực tiếp thông tin nhạy cảm (tên CSDL, username, password) vào mã nguồn, các thông tin này được lưu trữ trong file .env ở thư mục gốc. Trong file landifyapis/settings.py, các biến môi trường này được đọc và sử dụng để cấu hình mục DATABASES. Đặc biệt, ENGINE được thiết lập là 'django.contrib.gis.db.backends.postgis' để kích hoạt sự hỗ trợ cho extension PostGIS.

# landifyapis/settings.py (đoạn mã minh họa)

DATABASES = {

"default": {

"ENGINE": os.getenv('DB\_ENGINE', 'django.contrib.gis.db.backends.postgis'),

"NAME": os.getenv('DB\_NAME'),

"USER": os.getenv('DB\_USER'),

"PASSWORD": os.getenv('DB\_PASSWORD'),

"HOST": os.getenv('DB\_HOST', 'localhost'),

"PORT": os.getenv('DB\_PORT', '5432'),

}

}

Bước 2: Định nghĩa Lược đồ Dữ liệu qua Django Models

* + Toàn bộ cấu trúc của cơ sở dữ liệu PostgreSQL được định nghĩa hoàn toàn bằng các lớp Python trong các file models.py của từng app. Ví dụ, trong apps/listings/models.py, một thực thể Listing được định nghĩa như sau:

# apps/listings/models.py (đoạn mã minh họa)

class Listing(BaseModel):

property = models.ForeignKey(Property, on\_delete=models.CASCADE, ...)

user = models.ForeignKey(User, on\_delete=models.CASCADE, ...)

title = models.CharField(max\_length=255, ...)

price\_value = models.DecimalField(...)

# ... các trường khác

* + Django ORM sẽ tự động diễn dịch lớp Listing này thành một bảng có tên listings\_listing trong CSDL PostgreSQL, với các cột tương ứng (property\_id, user\_id, title, price\_value) và các kiểu dữ liệu phù hợp. Các mối quan hệ như ForeignKey (Nhiều-Một) hay ManyToManyField (Nhiều-Nhiều) cũng được định nghĩa trực tiếp tại đây.

Bước 3: Quản lý Thay đổi Lược đồ với Migrations

* + Dự án Landify tuân thủ nghiêm ngặt quy trình quản lý thay đổi CSDL bằng hệ thống "di trú" (migrations) của Django. Bất cứ khi nào có sự thay đổi trong file models.py (thêm/sửa/xóa một trường hoặc một model), hai lệnh sau sẽ được thực thi:
    1. python manage.py makemigrations: Django sẽ so sánh trạng thái hiện tại của các model với các file migration đã có và tự động tạo ra một file migration mới trong thư mục migrations/ của app tương ứng. File này chứa các chỉ dẫn bằng Python để áp dụng thay đổi.
    2. python manage.py migrate: Lệnh này sẽ thực thi các file migration chưa được áp dụng, chuyển đổi các chỉ dẫn Python thành các câu lệnh SQL (CREATE TABLE, ALTER TABLE...) và chạy chúng trên CSDL PostgreSQL.
  + Cách làm này đảm bảo rằng cấu trúc cơ sở dữ liệu luôn đồng bộ với mã nguồn và mọi thay đổi đều được ghi lại, giúp cho việc làm việc nhóm và triển khai lên các môi trường khác nhau trở nên an toàn và nhất quán.

Bước 4: Tương tác với Dữ liệu qua Django ORM

* + Trong toàn bộ mã nguồn của dự án, việc tương tác với PostgreSQL được thực hiện hoàn toàn thông qua Django ORM. Các ViewSet trong file views.py sử dụng các phương thức của ORM để truy vấn dữ liệu.
    1. Đọc dữ liệu (Read): Listing.objects.all() để lấy tất cả tin đăng, User.objects.get(pk=1) để lấy một người dùng cụ thể.
    2. Ghi dữ liệu (Create): Listing.objects.create(title='...', user=request.user, ...) để tạo một bản ghi mới.
    3. Cập nhật (Update): listing.save() sau khi đã thay đổi thuộc tính của một đối tượng.
    4. Xóa (Delete): listing.delete() để xóa một bản ghi.
* Bước 5: Tối ưu hóa Truy vấn
  + Để giải quyết vấn đề truy vấn N+1 và cải thiện hiệu năng, dự án đã áp dụng các kỹ thuật tối ưu hóa của Django ORM. Trong các ViewSet trả về danh sách, các phương thức select\_related và prefetch\_related được sử dụng rộng rãi.
    1. select\_related('user', 'property\_\_location'): Được sử dụng cho các mối quan hệ ForeignKey và OneToOneField. Nó sẽ thực hiện một câu lệnh JOIN trong SQL để lấy dữ liệu của các bảng liên quan trong cùng một lần truy vấn.
    2. prefetch\_related('feature\_values'): Được sử dụng cho các mối quan hệ ManyToManyField và ForeignKey ngược. Nó sẽ thực hiện một truy vấn riêng biệt để lấy tất cả các đối tượng liên quan và sau đó thực hiện việc "join" trong Python, giúp giảm số lượng truy vấn đến CSDL.

Bằng cách tận dụng tối đa sức mạnh của Django ORM và các công cụ đi kèm, việc triển khai và làm việc với PostgreSQL trong dự án Landify trở nên hiệu quả, an toàn và có cấu trúc rõ ràng, giúp các nhà phát triển có thể tập trung vào logic nghiệp vụ thay vì phải xử lý các câu lệnh SQL phức tạp.

#### Lý do lựa chọn PostgreSQL

Sự lựa chọn PostgreSQL làm hệ quản trị cơ sở dữ liệu chính cho Landify dựa trên những đặc tính kỹ thuật vượt trội, phù hợp với yêu cầu về độ tin cậy, hiệu năng và khả năng mở rộng của một nền tảng giao dịch.

Ưu điểm

Yêu cầu về Toàn vẹn Dữ liệu và Giao dịch An toàn:

* + Thách thức: Landify xử lý các nghiệp vụ phức tạp như tạo tin đăng hay xử lý thanh toán, đòi hỏi dữ liệu phải luôn ở trạng thái nhất quán, ngay cả khi có sự cố hệ thống xảy ra.
  + Giải pháp của PostgreSQL: PostgreSQL nổi tiếng là một trong những hệ quản trị CSDL mã nguồn mở ổn định và đáng tin cậy nhất. Nó tuân thủ đầy đủ các thuộc tính ACID (Atomicity, Consistency, Isolation, Durability), đảm bảo rằng các giao dịch (transactions) sẽ được thực thi một cách toàn vẹn. Đây là yếu tố sống còn để đảm bảo tính chính xác của dữ liệu trong một hệ thống giao dịch.

Nền tảng cho Hiệu năng và Khả năng Mở rộng trong Tương lai:

* + Thách thức: Hệ thống cần được thiết kế để xử lý các tải công việc lớn và các bộ dữ liệu phức tạp khi số lượng người dùng và tin đăng tăng lên.
  + Giải pháp của PostgreSQL: PostgreSQL cung cấp nhiều cơ chế tối ưu hóa hiệu năng tiên tiến như các loại chỉ mục (index) đa dạng (B-tree, GiST, GIN), tối ưu hóa truy vấn thông minh, và khả năng xử lý đồng thời nhiều kết nối hiệu quả thông qua cơ chế MVCC (Multi-Version Concurrency Control), sẵn sàng cho việc mở rộng quy mô.

Linh hoạt trong Lưu trữ Dữ liệu Phức tạp:

* + Thách thức: Một số dữ liệu trong ngành bất động sản không có cấu trúc cố định, ví dụ như các đặc điểm của một căn nhà có thể rất đa dạng.
  + Giải pháp của PostgreSQL: Vượt ra ngoài khuôn khổ của một CSDL quan hệ truyền thống, PostgreSQL hỗ trợ các kiểu dữ liệu hiện đại như JSONB (lưu trữ và truy vấn dữ liệu JSON hiệu quả). Trong Landify, kiểu dữ liệu JSONB đã được tận dụng trong model ListingPropertyFeatureValue để lưu trữ giá trị của các đặc điểm một cách linh hoạt mà không cần phải liên tục thay đổi cấu trúc bảng.

Khả năng Mở rộng Chức năng với Extensions:

* + Thách thức: Dự án có nhu cầu xử lý các dữ liệu không gian địa lý (geospatial data) để thực hiện các tính năng như tìm kiếm bất động sản theo vị trí, khoảng cách.
  + Giải pháp của PostgreSQL: Một trong những điểm mạnh lớn nhất của PostgreSQL là khả năng mở rộng thông qua các "extensions". Thay vì phải sử dụng một CSDL riêng biệt, extension PostGIS đã được cài đặt để bổ sung các kiểu dữ liệu và hàm xử lý không gian địa lý trực tiếp vào PostgreSQL. Đây là một yếu tố quyết định, giúp hệ thống trở nên mạnh mẽ và tinh gọn.

Nhược điểm

Độ phức tạp trong Cấu hình: So với MySQL hay SQLite, việc cấu hình và tối ưu hóa PostgreSQL ở quy mô lớn đòi hỏi kiến thức sâu hơn. Đây là một sự đánh đổi được chấp nhận để đổi lấy một hệ thống mạnh mẽ và giàu tính năng hơn.

Hiệu năng ghi đơn giản: Trong một số trường hợp rất cụ thể, PostgreSQL có thể có hiệu năng ghi thấp hơn một chút so với các giải pháp khác. Tuy nhiên, với một ứng dụng có cả đọc và ghi phức tạp như Landify, đặc tính này không phải là một vấn đề đáng kể.

## Các Kiến trúc và Công nghệ Nâng cao/Tối ưu hóa

### Kiến trúc Microservices - Chuyên môn hóa các Tác vụ Phức tạp:

#### Giới thiệu về Kiến trúc Microservices

Kiến trúc Microservices (Kiến trúc Vi dịch vụ) là một phương pháp thiết kế và phát triển phần mềm, trong đó một ứng dụng lớn và phức tạp được cấu thành từ một tập hợp các dịch vụ nhỏ, độc lập. Mỗi dịch vụ này được xây dựng để thực thi một nghiệp vụ kinh doanh cụ thể, chạy trong tiến trình riêng của nó, và giao tiếp với các dịch vụ khác thông qua các cơ chế gọn nhẹ, thường là qua các giao thức HTTP/API. Cách tiếp cận này đối lập hoàn toàn với kiến trúc nguyên khối (monolithic) truyền thống, nơi tất cả các chức năng được xây dựng và triển khai như một thể thống nhất duy nhất.

***Công dụng và Vai trò:***Trong một hệ thống lớn như Landify, không phải tất cả các tác vụ đều có cùng bản chất. Một số tác vụ, đặc biệt là những tác vụ liên quan đến Trí tuệ Nhân tạo (AI) hoặc xử lý tính toán nặng, thường đòi hỏi các thư viện chuyên biệt, môi trường vận hành riêng và có thể tiêu tốn nhiều tài nguyên hệ thống (CPU, RAM). Nếu tích hợp trực tiếp các tác vụ này vào ứng dụng Django chính, nó sẽ dẫn đến nhiều vấn đề:

* Làm "phình to" ứng dụng chính: Ứng dụng Django sẽ trở nên cồng kềnh, phức tạp và khó bảo trì.
* Xung đột thư viện: Các thư viện AI có thể yêu cầu các phiên bản phụ thuộc khác với những gì ứng dụng web đang sử dụng.
* Ảnh hưởng hiệu năng: Một tác vụ AI tiêu tốn nhiều CPU có thể làm chậm toàn bộ hệ thống API, ảnh hưởng đến trải nghiệm của tất cả người dùng.

Kiến trúc Microservices được áp dụng trong Landify để giải quyết triệt để các vấn đề này. Vai trò của nó là tách biệt và chuyên môn hóa các tác vụ phức tạp ra khỏi hệ thống backend chính. Mỗi microservice hoạt động như một "chuyên gia", chỉ làm một việc duy nhất và làm thật tốt.

***Ưu điểm và Lý do áp dụng:***Việc áp dụng kiến trúc Microservices mang lại nhiều lợi ích chiến lược cho sự phát triển và vận hành của Landify.

* Tính Độc lập và Linh hoạt (Independence & Flexibility): Mỗi microservice có thể được phát triển, triển khai, và mở rộng quy mô một cách hoàn toàn độc lập với ứng dụng chính và các microservice khác. Điều này cho phép lựa chọn công nghệ phù hợp nhất cho từng bài toán cụ thể. Ví dụ, một microservice AI có thể được xây dựng bằng FastAPI (một framework Python gọn nhẹ) để tối ưu cho tốc độ, trong khi hệ thống chính vẫn sử dụng Django để tận dụng hệ sinh thái toàn diện của nó.
* Tăng cường Khả năng Bảo trì và Nâng cấp (Improved Maintainability): Mã nguồn của mỗi microservice nhỏ hơn, tập trung hơn, giúp cho việc hiểu, bảo trì và nâng cấp trở nên dễ dàng hơn rất nhiều. Nếu cần thay đổi thuật toán trong microservice xác thực khuôn mặt, nhóm phát triển có thể làm điều đó mà không cần phải đụng đến hay triển khai lại toàn bộ ứng dụng Django.
* Nâng cao Độ tin cậy và Khả năng chịu lỗi (Enhanced Reliability): Sự cố xảy ra ở một microservice sẽ không làm sập toàn bộ hệ thống. Ví dụ, nếu microservice chatbot phong thủy tạm thời gặp lỗi, các chức năng cốt lõi khác của Landify như đăng tin, tìm kiếm, chat vẫn hoạt động bình thường.
* Tối ưu hóa Hiệu năng và Tài nguyên: Các microservice có thể được triển khai trên các máy chủ riêng biệt và được tối ưu hóa tài nguyên theo đúng nhu cầu của chúng. Một microservice xử lý AI có thể được chạy trên một máy chủ có GPU mạnh mẽ, trong khi ứng dụng Django chính chạy trên các máy chủ được tối ưu hóa cho các tác vụ I/O.

Khuyết điểm:

* Tăng độ phức tạp trong Vận hành (Operational Complexity): Việc quản lý, triển khai và giám sát nhiều dịch vụ riêng lẻ sẽ phức tạp hơn so với một ứng dụng nguyên khối. Nó đòi hỏi các công cụ và quy trình vận hành (DevOps) chuyên nghiệp hơn.
* Độ trễ Mạng (Network Latency): Việc giao tiếp giữa các dịch vụ thông qua mạng có thể tạo ra độ trễ, đòi hỏi phải có các cơ chế xử lý lỗi mạng và retry một cách cẩn thận.
* Tính nhất quán Dữ liệu Phân tán (Distributed Data Consistency): Việc đảm bảo tính nhất quán của dữ liệu trên nhiều dịch vụ khác nhau là một thách thức kỹ thuật phức tạp.

#### Cách triển khai Kiến trúc Microservices trong dự án Landify

Việc hiện thực hóa kiến trúc microservices cho Landify không chỉ dừng lại ở việc phát triển mã nguồn cho từng dịch vụ riêng lẻ, mà còn bao gồm một quy trình triển khai và vận hành có hệ thống để đảm bảo các dịch vụ này có thể hoạt động một cách độc lập, ổn định và giao tiếp được với nhau. Quá trình này dựa trên hai công nghệ nền tảng là Docker để đóng gói và Hugging Face Spaces để lưu trữ và vận hành.

***Tổng quan các bước triển khai:***

1. Lựa chọn Công nghệ Nền tảng cho Microservices: Chọn một web framework gọn nhẹ và hiệu năng cao để xây dựng API cho từng dịch vụ.
2. Containerization (Đóng gói) với Docker: Đóng gói mỗi microservice, cùng với tất cả các thư viện và phụ thuộc của nó, vào một môi trường cô lập và có thể tái tạo gọi là Docker container.
3. Triển khai và Vận hành trên Hugging Face Spaces: Sử dụng nền tảng Hugging Face Spaces để lưu trữ, xây dựng (build) và chạy các Docker container này, cung cấp một endpoint API công khai.
4. Tích hợp vào Hệ thống Django Chính: Cấu hình ứng dụng Django để giao tiếp với các endpoint API của các microservice đã được triển khai.

***Chi tiết triển khai trong Landify:***

* Bước 1: Lựa chọn FastAPI làm Framework Nền tảng
  + Để đảm bảo các microservice có hiệu năng cao và thời gian phản hồi nhanh, FastAPI đã được lựa chọn làm web framework chính. Khác với Django, FastAPI được thiết kế tối ưu cho việc xây dựng API, có tốc độ xử lý vượt trội và hỗ trợ lập trình bất đồng bộ, rất phù hợp cho các tác vụ AI cần xử lý nhanh. Hơn nữa, khả năng tự động sinh tài liệu API của FastAPI cũng giúp cho việc kiểm thử và tích hợp trở nên dễ dàng hơn.
* **Bước 2: Containerization (Đóng gói) với Docker**
  + Để giải quyết vấn đề "chạy được trên máy tôi nhưng không chạy được trên máy bạn" và đảm bảo tính nhất quán giữa môi trường phát triển và sản phẩm, mỗi microservice đều được đóng gói bằng Docker.
  + Một file Dockerfile được tạo ra cho mỗi microservice. File này chứa một bộ các chỉ dẫn từng bước để xây dựng một "image" – một bản mẫu hoàn chỉnh của môi trường chạy.

**Ví dụ về một Dockerfile tiêu biểu cho microservice OCR:**

# Sử dụng một ảnh base Python 3.8 chính thức

FROM python:3.8

# Thiết lập thư mục làm việc bên trong container

WORKDIR /code

# Cài đặt các phụ thuộc hệ thống nếu cần (ví dụ: thư viện xử lý ảnh)

RUN apt-get update && apt-get install -y libgomp1 && rm -rf /var/lib/apt/lists/\*

# Sao chép file requirements.txt vào trước để tận dụng cache của Docker

COPY ./requirements.txt /code/requirements.txt

# Cài đặt các thư viện Python từ requirements.txt

RUN pip install --no-cache-dir -r /code/requirements.txt

# Sao chép toàn bộ code của ứng dụng vào container

COPY . /code/

# Mở cổng mà ứng dụng FastAPI sẽ lắng nghe

EXPOSE 7860

# Lệnh để khởi động ứng dụng FastAPI khi container chạy

CMD ["uvicorn", "app:app", "--host", "0.0.0.0", "--port", "7860"]

* + File Dockerfile này đảm bảo rằng bất kể được chạy ở đâu, microservice cũng sẽ có cùng một phiên bản Python, cùng một bộ thư viện với phiên bản chính xác, tạo ra một môi trường vận hành có thể dự đoán và ổn định.
* Bước 3: Triển khai và Vận hành trên Hugging Face Spaces
  + Thay vì phải tự mình thiết lập và quản lý các máy chủ phức tạp, dự án đã tận dụng nền tảng **Hugging Face Spaces** để triển khai các microservice AI một cách nhanh chóng và hiệu quả. Hugging Face Spaces là một dịch vụ cho phép các nhà phát triển lưu trữ, chia sẻ và chạy các mô hình AI và ứng dụng web một cách công khai.
  + Quy trình triển khai:
    1. Mã nguồn của mỗi microservice, bao gồm cả Dockerfile và file requirements.txt, được đẩy lên một kho chứa (repository) riêng trên Hugging Face.
    2. Một file cấu hình README.md đặc biệt được tạo ra để thông báo cho nền tảng biết cách xây dựng và chạy ứng dụng.

Ví dụ về file README.md cấu hình cho Hugging Face Spaces:

---

title: Vietnamese Citizen ID OCR

sdk: docker

app\_port: 7860

---

* + 1. sdk: docker: Chỉ thị cho Hugging Face sử dụng Dockerfile có trong repository để xây dựng image.
    2. app\_port: 7860: Thông báo rằng ứng dụng bên trong container sẽ chạy ở cổng 7860, để nền tảng có thể ánh xạ (map) cổng này ra ngoài internet.
  + Sau khi được đẩy lên, Hugging Face Spaces sẽ tự động thực hiện quá trình xây dựng Docker image và khởi chạy container. Mỗi microservice sẽ được cấp một URL công khai (ví dụ: https://ten-cua-ban-landify-cccd-ocr.hf.space), sẵn sàng để nhận các yêu cầu API.
* **Bước 4: Tích hợp vào Hệ thống Django Chính**
  + Cuối cùng, các URL công khai của các microservice trên Hugging Face Spaces được cấu hình trong file .env của dự án Django chính.
  + Trong các file services.py của Django (ví dụ: apps.verification/services.py), các hàm sẽ sử dụng thư viện requests để gửi các yêu cầu HTTP đến các URL này. Từ góc nhìn của ứng dụng Django, việc gọi đến một microservice tự xây dựng cũng tương tự như việc gọi đến một API của bất kỳ bên thứ ba nào khác, hoàn thành việc tích hợp một cách liền mạch và phân tách rõ ràng.

Cách tiếp cận này đã cho phép Landify tận dụng được sức mạnh của kiến trúc microservices—tính độc lập, chuyên môn hóa—mà không cần phải gánh chịu gánh nặng về việc tự quản lý hạ tầng phức tạp, giúp đẩy nhanh quá trình phát triển và triển khai sản phẩm.

#### Lý do áp dụng Kiến trúc Microservices cho dự án Landify

Quyết định áp dụng kiến trúc Microservices, thay vì tích hợp tất cả các chức năng vào ứng dụng Django nguyên khối, là một quyết định kiến-trúc chiến lược. Lựa chọn này xuất phát từ việc phân tích sâu sắc các yêu cầu kỹ thuật đặc thù của dự án và tầm nhìn phát triển dài hạn, nhằm giải quyết các bài toán về hiệu năng, khả năng bảo trì và sự linh hoạt của công nghệ.

* Chuyên môn hóa và Tối ưu hóa Công nghệ cho từng Tác vụ:
  + Lý do: Các chức năng trong hệ thống Landify có bản chất rất khác nhau. Trong khi các nghiệp vụ CRUD (Tạo, Đọc, Cập nhật, Xóa) thông thường phù hợp với hệ sinh thái toàn diện của Django, thì các tác vụ xử lý AI như nhận dạng ký tự (OCR) hay so khớp khuôn mặt lại đòi hỏi các thư viện chuyên biệt (vietocr, face\_recognition) và một môi trường được tối ưu hóa cho tốc độ tính toán. Việc ép buộc tất cả vào cùng một ứng dụng sẽ tạo ra một hệ thống cồng kềnh và không hiệu quả.
  + Trong đề tài: Bằng cách tách các tác vụ AI ra thành các microservice riêng biệt được xây dựng bằng FastAPI, dự án đã có thể lựa chọn công nghệ tốt nhất cho từng công việc. FastAPI, với hiệu năng vượt trội và hỗ trợ lập trình bất đồng bộ, là lựa chọn lý tưởng để xây dựng các API AI cần phản hồi nhanh. Trong khi đó, ứng dụng Django chính vẫn tận dụng được thế mạnh của mình về ORM, hệ thống quản trị và các tính năng web toàn diện. Điều này cho phép mỗi thành phần của hệ thống được tối ưu hóa cho đúng nhiệm-vụ của nó.
* Đảm bảo Hiệu năng và Khả năng Mở rộng của Hệ thống Chính:
  + Lý do: Các tác vụ AI thường tiêu tốn rất nhiều tài nguyên hệ thống, đặc biệt là CPU và bộ nhớ. Nếu các tác vụ này chạy cùng tiến trình với các API web thông thường, một request xử lý ảnh hoặc video nặng có thể làm "treo" hoặc làm chậm đáng kể toàn bộ hệ thống, ảnh hưởng đến trải nghiệm của tất cả người dùng khác đang thực hiện các thao tác đơn giản như tìm kiếm hay xem tin đăng.
  + Trong đề tài: Kiến trúc microservices đã giải quyết triệt để vấn đề này. Các tác vụ nặng được "cô lập" và chạy trên các tài nguyên riêng (trong trường hợp này là trên hạ tầng của Hugging Face Spaces). Ứng dụng Django chính chỉ thực hiện một lời gọi HTTP gọn nhẹ và chờ kết quả. Điều này đảm bảo rằng dù cho microservice AI có đang xử lý một tác vụ phức tạp đến đâu, hiệu năng của các API cốt lõi trong Landify vẫn được duy trì ổn định, mang lại trải nghiệm người dùng mượt mà và đáng tin cậy.
* Tăng cường Khả năng Bảo trì và Nâng cấp Độc lập:
  + Lý do: Thế giới AI và học máy phát triển với tốc độ chóng mặt. Một thư viện hoặc mô hình tốt nhất hôm nay có thể trở nên lỗi thời vào ngày mai. Trong một kiến trúc nguyên khối, việc nâng cấp một thư viện AI có thể gây ra xung đột phụ thuộc (dependency hell) và ảnh hưởng đến toàn bộ ứng dụng, khiến cho việc cập nhật trở nên rủi ro và tốn kém.
  + Trong đề tài: Với kiến trúc microservices, mỗi dịch vụ là một thực thể độc lập với vòng đời riêng. Nhóm phát triển có thể tự do nâng cấp, sửa lỗi, hoặc thậm chí là viết lại hoàn toàn một microservice (ví dụ: thay thế thư viện face\_recognition bằng một giải pháp thương mại tốt hơn) mà không cần phải đụng đến một dòng code nào trong ứng dụng Django chính. Sự độc lập này giúp cho hệ thống Landify trở nên cực kỳ linh hoạt, dễ dàng thích ứng với sự thay đổi của công nghệ và kéo dài tuổi thọ của sản phẩm.
* Đơn giản hóa Quá trình Triển khai và Vận hành:
  + Lý do: Việc thiết lập một môi trường thống nhất để chạy cả ứng dụng web và các thư viện AI phức tạp có thể là một thách thức lớn. Các thư viện AI thường có những yêu cầu rất đặc thù về môi trường hệ điều hành và các gói phụ thuộc.
  + Trong đề tài: Bằng cách sử dụng Docker để đóng gói từng microservice, dự án đã loại bỏ hoàn toàn các vấn đề về tương thích môi trường. Hơn nữa, việc tận dụng nền tảng Hugging Face Spaces đã giúp đơn giản hóa đáng kể quá trình triển khai. Thay vì phải tự mình cấu hình máy chủ, quản lý tài nguyên và bảo mật, nhóm có thể tập trung hoàn toàn vào việc phát triển mã nguồn, trong khi việc vận hành đã được một nền tảng chuyên nghiệp đảm nhiệm.

Tóm lại, việc áp dụng kiến trúc microservices không chỉ là một giải pháp kỹ thuật, mà còn là một quyết định chiến lược giúp Landify trở nên mạnh mẽ, linh hoạt và dễ quản lý hơn. Nó cho phép hệ thống vừa có thể xử lý các nghiệp vụ web truyền thống một cách hiệu quả, vừa có thể tích hợp các công nghệ AI tiên tiến mà không làm ảnh hưởng đến sự ổn định và hiệu năng chung.

### Microservice Trích xuất Thông tin CCCD (OCR)

Microservice Trích xuất Thông tin CCCD là một dịch vụ độc lập, chuyên biệt được xây dựng để giải quyết bài toán nhận dạng và "đọc" dữ liệu từ hình ảnh Căn cước công dân (CCCD) gắn chip của Việt Nam. Đây là thành phần công nghệ cốt lõi, đóng vai trò là bước đầu tiên trong quy trình xác thực danh tính điện tử (eKYC) của hệ thống Landify.

***Công dụng và Vai trò:***Vai trò chính của microservice này là tự động hóa quá trình nhập liệu thông tin cá nhân, giảm thiểu sai sót do người dùng tự nhập và cung cấp một nguồn dữ liệu đáng tin cậy cho việc xác thực ở các bước tiếp theo. Cụ thể, dịch vụ này thực hiện hai nhiệm vụ chính:

* Nhận dạng Ký tự Quang học (Optical Character Recognition - OCR): Dịch vụ nhận đầu vào là một tệp hình ảnh (mặt trước của CCCD) và sử dụng các mô hình học máy để định vị, nhận dạng và chuyển đổi các vùng văn bản trên ảnh thành dạng chuỗi ký tự có thể đọc được bằng máy tính. Các trường thông tin được trích xuất bao gồm: Số CCCD, Họ và tên, Ngày sinh, Giới tính, Quốc tịch, Quê quán, và Nơi thường trú.
* Phát hiện và Cắt ảnh Chân dung (Face Detection & Cropping): Đồng thời với việc đọc chữ, microservice cũng có khả năng phát hiện vùng chứa ảnh chân dung trên CCCD, tự động cắt (crop) và tách riêng phần ảnh này ra. Ảnh chân dung sau khi được cắt sẽ được mã hóa dưới dạng chuỗi base64 và trả về cùng với các thông tin văn bản. Dữ liệu này là đầu vào cực kỳ quan trọng cho microservice Xác thực Khuôn mặt ở bước sau.

Công nghệ Nền tảng:  
Để đạt được độ chính xác cao khi xử lý ảnh CCCD tiếng Việt có dấu, microservice này được xây dựng dựa trên sự kết hợp của các thư viện mã nguồn mở hàng đầu trong lĩnh vực thị giác máy tính:

* FastAPI: Là web framework được sử dụng để xây dựng API cho microservice, đảm bảo hiệu năng cao và thời gian phản hồi nhanh.
* PaddleOCR: Một bộ công cụ OCR mạnh mẽ, được sử dụng trong giai đoạn đầu để phát hiện vị trí của các dòng văn bản (text detection) trên toàn bộ ảnh CCCD.
* VietOCR: Là thư viện OCR chuyên biệt cho tiếng Việt, được xây dựng dựa trên các kiến trúc mô hình học sâu tiên tiến. Sau khi PaddleOCR xác định được các vùng chữ, VietOCR sẽ chịu trách nhiệm "đọc" và nhận dạng nội dung của từng vùng đó (text recognition) với độ chính xác cao, kể cả với các ký tự có dấu.
* OpenCV: Là thư viện thị giác máy tính kinh điển, được sử dụng để thực hiện các tác vụ tiền xử lý ảnh và đặc biệt là để chạy mô hình haarcascade nhằm phát hiện và định vị chính xác vùng ảnh chân dung trên CCCD.

#### Cách triển khai Microservice Trích xuất Thông tin CCCD (OCR)

Việc xây dựng một microservice có khả năng "đọc" và hiểu được một tài liệu phức tạp như Căn cước công dân (CCCD) đòi hỏi một quy trình kỹ thuật gồm nhiều bước, từ lựa chọn công cụ, xử lý dữ liệu, cho đến việc huấn luyện và tối ưu hóa mô hình Trí tuệ Nhân tạo.

***Tổng quan các bước triển khai:***

1. Lựa chọn và Tích hợp các Thư viện Nền tảng: Xây dựng một môi trường Python với các thư viện chuyên dụng cho xử lý ảnh và nhận dạng ký tự quang học.
2. Phân tích Thách thức và Phát triển Giải pháp: Xác định những khó khăn đặc thù khi nhận dạng CCCD và đề ra một chiến lược xử lý đa tầng thay vì chỉ dựa vào một mô hình OCR duy nhất.
3. Thu thập và Đánh nhãn Dữ liệu: Xây dựng một bộ dữ liệu (dataset) tùy chỉnh từ các ảnh CCCD mẫu và sử dụng các công cụ chuyên dụng để đánh nhãn các vùng thông tin.
4. Huấn luyện và Tinh chỉnh Mô hình: Sử dụng bộ dữ liệu đã đánh nhãn để huấn luyện các mô hình học máy, giúp chúng "học" cách nhận dạng chính xác các trường thông tin trong bối cảnh phức tạp của CCCD.
5. Đóng gói và Triển khai: Hoàn thiện microservice dưới dạng một API độc lập và triển khai lên môi trường hoạt động.

***Chi tiết triển khai trong Landify:***

* Bước 1: Lựa chọn và Tích hợp Thư viện Nền tảng
  + Môi trường của microservice được xây dựng dựa trên một bộ các thư viện Python mạnh mẽ, được cài đặt thông qua file requirements.txt. Các thư viện chính bao gồm:
    - FastAPI: Được chọn làm web framework để xây dựng API nhờ hiệu năng cao và khả năng phát triển nhanh.
    - PaddleOCR và PaddlePaddle: Là bộ công cụ OCR mã nguồn mở được phát triển bởi Baidu. PaddleOCR được sử dụng cho nhiệm vụ phát hiện văn bản (Text Detection), tức là xác định các vùng chữ nhật (bounding box) bao quanh các cụm từ trên ảnh.
    - VietOCR và PyTorch: VietOCR là một thư viện OCR tiên tiến, được tối ưu hóa đặc biệt cho tiếng Việt. Nó được sử dụng cho nhiệm vụ nhận dạng văn bản (Text Recognition), "đọc" nội dung bên trong các vùng chữ nhật do PaddleOCR xác định.
    - OpenCV: Là thư viện nền tảng cho hầu hết các tác vụ xử lý ảnh, từ việc đọc file, tiền xử lý (chuyển ảnh xám, tăng độ tương phản) cho đến việc chạy các mô hình phát hiện đối tượng như haarcascade để tìm ảnh chân dung.
* Bước 2: Phân tích Thách thức và Phát triển Giải pháp Xử lý Đa tầng
  + Khó khăn gặp phải: Trong quá trình thử nghiệm ban đầu, nhóm nhận thấy rằng việc áp dụng trực tiếp mô hình VietOCR (vốn được huấn luyện rất kỹ trên các bộ dữ liệu văn bản thông thường) lên toàn bộ ảnh CCCD cho kết quả rất thấp. Nguyên nhân là do mô hình này được tối ưu để nhận dạng các dòng văn bản đơn lẻ, trong khi ảnh CCCD có những đặc điểm cực kỳ phức tạp:
    - Bố cục đa dòng và đa cột: Các thông tin như "Họ và tên" hay "Nơi thường trú" thường bị xuống dòng.
    - Nhiều loại phông chữ và màu sắc: Chữ in hoa, in thường, chữ đỏ (số CCCD) xuất hiện lẫn lộn.
    - Nền ảnh phức tạp: Các họa tiết chìm, hình ảnh quốc huy và con dấu gây nhiễu cho quá trình nhận dạng.
    - Song ngữ Anh/Việt: Sự hiện diện của cả hai ngôn ngữ làm tăng độ phức tạp.
  + Giải pháp đề xuất: Nhận thấy không thể giải quyết bài toán chỉ bằng một bước, một pipeline xử lý đa tầng đã được thiết kế:
    - Phát hiện vùng văn bản (Detection): Dùng PaddleOCR để xác định vị trí của tất cả các cụm từ trên ảnh.
    - Nhận dạng từng cụm (Recognition): Dùng VietOCR để đọc nội dung của từng vùng chữ nhật nhỏ đã được phát hiện.
    - Phân loại và Gán nhãn (Classification & Extraction): Đây là bước quan trọng nhất. Sau khi đã có một danh sách các cụm từ và vị trí của chúng, hệ thống cần một "bộ não" để hiểu được "cụm từ này thuộc về trường thông tin nào?". Để làm được điều này, cần phải huấn luyện một mô hình tùy chỉnh.
* Bước 3: Thu thập và Đánh nhãn Dữ liệu
  + Một bộ dữ liệu gồm các ảnh CCCD mẫu đã được thu thập.
  + Để huấn luyện mô hình phân loại và gán nhãn, việc đánh nhãn (labeling) dữ liệu là bắt buộc. Nhóm đã sử dụng các công cụ đánh nhãn chuyên dụng như VietOCR Labeling Tool (hoặc bạn có thể sử dụng nền tảng như DORI để thực hiện công việc này).
  + Quá trình đánh nhãn bao gồm việc vẽ các hộp giới hạn (bounding box) xung quanh từng trường thông tin (ví dụ: toàn bộ vùng "Họ và tên") và gán cho nó một nhãn tương ứng (ví dụ: Name). Dữ liệu đầu ra của quá trình này là một tập hợp các ảnh và các file chú thích (annotation files) mô tả vị trí và nhãn của từng trường thông tin.
* Bước 4: Huấn luyện và Tinh chỉnh Mô hình
  + Bộ dữ liệu đã được đánh nhãn sau đó được sử dụng để huấn luyện một mô hình tùy chỉnh. Các kỹ thuật học sâu (Deep Learning), cụ thể là các kiến trúc mạng nơ-ron tích chập (Convolutional Neural Networks - CNN) hoặc các mô hình dựa trên Transformer, được áp dụng.
  + Quá trình huấn luyện giúp mô hình "học" được mối liên hệ giữa vị trí tương đối của các cụm từ và nhãn ngữ nghĩa của chúng. Ví dụ, nó học được rằng cụm từ nằm bên phải của chuỗi "Họ và tên/Full Name" chính là tên của người đó.
  + Sau khi huấn luyện, mô hình này được tích hợp vào microservice để thực hiện bước phân loại và gán nhãn tự động.

Bằng cách kết hợp sức mạnh của các mô hình OCR có sẵn và việc huấn luyện một mô hình tùy chỉnh để "hiểu" được bố cục của CCCD, microservice OCR của Landify đã vượt qua được những thách thức ban đầu và đạt được độ chính xác cao trong việc trích xuất thông tin một cách hoàn toàn tự động.

### Microservice Xác thực Khuôn mặt

Microservice Xác thực Khuôn mặt là một dịch vụ chuyên biệt, đóng vai trò là bước thứ hai và cũng là bước cuối cùng trong quy trình xác thực danh tính điện tử (eKYC) của hệ thống Landify. Dịch vụ này được xây dựng để giải quyết một bài toán cốt lõi về bảo mật và chống gian lận: xác minh rằng người đang thực hiện quy trình là một người thật và chính là chủ nhân của giấy tờ tùy thân đã cung cấp ở bước trước.

***Công dụng và Vai trò:***Vai trò chính của microservice này là thực hiện so khớp sinh trắc học (biometric matching) giữa hai nguồn dữ liệu khác nhau: ảnh chân dung tĩnh và một đoạn video động. Cụ thể, dịch vụ này đảm nhiệm hai nhiệm vụ quan trọng:

* Xác thực Sự sống (Liveness Detection): Đây là một kỹ thuật chống giả mạo (anti-spoofing) tiên tiến. Thay vì chỉ so sánh hai ảnh tĩnh, microservice có khả năng phân tích một đoạn video ngắn do người dùng quay để xác định xem khuôn mặt trong video có phải là của một người thật, đang sống hay không. Điều này giúp ngăn chặn các hình thức gian lận tinh vi như sử dụng ảnh in, ảnh trên màn hình điện thoại, hoặc mặt nạ để vượt qua hệ thống xác thực.
* So khớp Khuôn mặt (Face Matching/Verification): Sau khi đã xác nhận "sự sống", dịch vụ sẽ tiến hành so sánh khuôn mặt được phát hiện trong video với ảnh chân dung đã được trích xuất từ CCCD ở microservice OCR. Quá trình này không chỉ là một phép so sánh hình ảnh đơn thuần, mà là một phép đối chiếu toán học giữa các vector đặc trưng sinh trắc học của hai khuôn mặt. Kết quả của quá trình này là một quyết định "khớp" hoặc "không khớp", cùng với một điểm số tương đồng (similarity score).

Công nghệ Nền tảng:  
Để thực hiện các tác vụ so khớp sinh trắc học phức tạp, microservice này được xây dựng dựa trên các thư viện và mô hình học sâu chuyên dụng trong lĩnh vực thị giác máy tính:

* FastAPI: Tương tự như microservice OCR, FastAPI được chọn làm web framework để đảm bảo API có hiệu năng cao và độ trễ thấp, yếu tố cực kỳ quan trọng trong một quy trình xác thực cần phản hồi nhanh cho người dùng.
* face\_recognition: Đây là thư viện cốt lõi của microservice. Nó được xây dựng dựa trên các thuật toán học sâu tiên tiến từ thư viện dlib, cung cấp các chức năng nền tảng:
  + Phát hiện Khuôn mặt (Face Detection): Định vị vị trí của các khuôn mặt trong từng khung hình của video.
  + Mã hóa Khuôn mặt (Face Encoding): Chuyển đổi mỗi khuôn mặt được phát hiện thành một vector đặc trưng 128 chiều. Vector này là một biểu diễn toán học duy nhất cho các đặc điểm sinh trắc học của khuôn mặt đó.
  + So sánh Khuôn mặt (Face Comparison): Tính toán khoảng cách Euclidean giữa hai vector đặc trưng. Khoảng cách càng nhỏ, hai khuôn mặt càng giống nhau.
* OpenCV và Pillow: Hai thư viện này đóng vai trò là công cụ phụ trợ, chịu trách nhiệm đọc và xử lý các tệp hình ảnh (được giải mã từ chuỗi base64) và các khung hình (frames) được trích xuất từ tệp video đầu vào.

#### Cách triển khai Microservice Xác thực Khuôn mặt

Việc xây dựng một microservice có khả năng so khớp khuôn mặt một cách chính xác và hiệu quả trong điều kiện thực tế là một bài toán đầy thách thức. Quá trình này không chỉ đơn thuần là việc gọi một vài hàm từ thư viện, mà đòi hỏi sự tối ưu hóa ở nhiều khâu, từ xử lý dữ liệu đầu vào cho đến việc tinh chỉnh thuật toán.

***Tổng quan các bước triển khai:***

1. Thiết lập Môi trường và Lựa chọn Thư viện: Xây dựng một môi trường Python độc lập và lựa chọn thư viện nhận dạng khuôn mặt phù hợp.
2. Thiết kế Giao diện API: Định nghĩa một endpoint API có khả năng nhận đồng thời cả dữ liệu hình ảnh (từ CCCD) và dữ liệu video (từ người dùng).
3. Phân tích Thách thức và Phát triển Thuật toán Xử lý: Xây dựng một pipeline xử lý video để giải quyết các vấn đề về hiệu năng và độ chính xác trong điều kiện thực tế.
4. Hiện thực hóa Logic So khớp và Trả về Kết quả: Viết mã nguồn để thực thi pipeline xử lý và trả về một kết quả xác thực rõ ràng.
5. Đóng gói và Triển khai: Sử dụng Docker và Hugging Face Spaces để triển khai microservice.

***Chi tiết triển khai trong Landify:***

* Bước 1: Thiết lập Môi trường và Lựa chọn Thư viện face-recognition
  + Môi trường cho microservice này được xây dựng độc lập, với các thư viện chính được liệt kê trong file requirements.txt.
  + Thư viện cốt lõi face-recognition được lựa chọn vì sự đơn giản trong việc sử dụng nhưng vẫn dựa trên các mô hình học sâu mạnh mẽ của dlib. Tuy nhiên, việc cài đặt dlib thường gặp nhiều khó khăn do yêu cầu phải biên dịch từ mã nguồn C++. Để giải quyết vấn đề này, dự án đã sử dụng một tệp wheel (.whl) đã được biên dịch sẵn cho môi trường Linux (phù hợp với môi trường triển khai Docker), giúp quá trình cài đặt trở nên nhanh chóng và đáng tin cậy.
* Bước 2: Thiết kế Giao diện API với FastAPI
  + Một ứng dụng FastAPI được tạo ra với một endpoint duy nhất: POST /verify/.
  + Thách thức: Endpoint này cần nhận đồng thời hai loại dữ liệu có định dạng khác nhau: một tệp video (dữ liệu nhị phân) và một chuỗi JSON chứa ảnh chân dung đã được mã hóa base64 (dữ liệu văn bản).
  + Giải pháp: FastAPI cung cấp một giải pháp thanh lịch cho vấn đề này bằng cách sử dụng File và Form. Endpoint được định nghĩa để nhận video\_file: UploadFile = File(...) và json\_data: str = Form(...). Điều này cho phép client có thể gửi một request multipart/form-data, chứa cả hai loại dữ liệu một cách dễ dàng.
* Bước 3: Phân tích Thách thức và Phát triển Thuật toán Xử lý Video
  + Khó khăn gặp phải: Thách thức lớn nhất khi làm việc với video là vấn đề hiệu năng. Một đoạn video ngắn vài giây có thể chứa hàng trăm khung hình. Việc xử lý (phát hiện và mã hóa khuôn mặt) trên tất cả các khung hình sẽ cực kỳ tốn thời gian và tài nguyên CPU, dẫn đến thời gian phản hồi API rất lâu, gây trải nghiệm người dùng kém. Hơn nữa, không phải khung hình nào cũng có chất lượng tốt; một số có thể bị mờ, bị che khuất, hoặc không có khuôn mặt.
  + Giải pháp - Pipeline Tối ưu hóa: Để giải quyết các vấn đề trên, một pipeline xử lý đã được thiết kế và tối ưu hóa:
    1. Xử lý "Khuôn mặt đã biết" (Known Face): Đầu tiên, chuỗi JSON được phân tích để lấy ra ảnh chân dung dạng base64. Ảnh này được giải mã và đưa qua mô hình face\_recognition một lần duy nhất để tính toán và lưu lại vector đặc trưng 128 chiều của "khuôn mặt đã biết".
    2. Lấy mẫu Khung hình (Frame Sampling): Thay vì xử lý mọi khung hình, microservice chỉ xử lý một khung hình sau mỗi VIDEO\_FRAME\_PROCESS\_RATE khung hình (ví dụ: chỉ xử lý khung hình thứ 15, 30, 45...). Kỹ thuật lấy mẫu này giúp giảm đáng kể khối lượng tính toán mà vẫn đảm bảo bao quát được nội dung của video.
    3. Giảm kích thước Khung hình (Frame Resizing): Trước khi đưa vào mô hình nhận dạng, mỗi khung hình được chọn sẽ được giảm kích thước xuống một chiều rộng cố định (RESIZE\_FRAME\_WIDTH, ví dụ: 320 pixels). Việc này giúp tăng tốc độ xử lý của mô hình AI một cách đáng kể vì mô hình sẽ phải làm việc trên một ma trận điểm ảnh nhỏ hơn, trong khi vẫn giữ lại đủ thông tin cần thiết để nhận dạng khuôn mặt.
    4. Tìm kiếm Khoảng cách Tốt nhất (Best Match Search): Dịch vụ không dừng lại ở lần so khớp đầu tiên. Nó sẽ lặp qua tất cả các khung hình đã được lấy mẫu, tính toán khoảng cách Euclidean giữa "khuôn mặt đã biết" và tất cả các khuôn mặt tìm thấy trong khung hình đó. Nó sẽ liên tục theo dõi và lưu lại khoảng cách nhỏnhất (best match distance) tìm được trong toàn bộ video.
* Bước 4: Hiện thực hóa Logic So khớp và Trả về Kết quả
  + Sau khi đã xử lý hết video và có được best\_match\_distance, logic so khớp cuối cùng được thực hiện.
  + Một ngưỡng (threshold) được định nghĩa trước, gọi là FACE\_MATCH\_TOLERANCE (ví dụ: 0.6). Nếu best\_match\_distance nhỏ hơn hoặc bằng ngưỡng này, hai khuôn mặt được coi là trùng khớp (verified: true). Ngược lại, chúng được coi là không khớp (verified: false).
  + Ngoài kết quả boolean, microservice còn tính toán và trả về một điểm số tương đồng theo thang phần trăm (face\_similarity\_percent) để cung cấp thêm thông tin chi tiết.
  + Xử lý Lỗi: Một trường hợp đặc biệt được xử lý là khi video không chứa bất kỳ khuôn mặt nào. Trong trường hợp này, dịch vụ sẽ trả về kết quả thất bại với một thông báo lỗi rõ ràng.

Bằng cách áp dụng một pipeline xử lý được tối ưu hóa, đặc biệt là các kỹ thuật lấy mẫu và giảm kích thước khung hình, microservice Xác thực Khuôn mặt của Landify đã giải quyết thành công bài toán về hiệu năng khi xử lý video. Giải pháp này không chỉ đảm bảo độ chính xác trong việc so khớp mà còn giữ được thời gian phản hồi của API ở mức chấp nhận được, tạo ra một quy trình eKYC vừa an toàn vừa hiệu quả.

### Celery và Redis

Trong một ứng dụng web phức tạp như Landify, không phải tất cả các công việc đều cần được thực hiện ngay lập tức trong luồng yêu cầu-phản hồi chính. Một số tác vụ có thể tốn nhiều thời gian, chẳng hạn như gửi thông báo đến nhiều người dùng, xử lý hình ảnh, hoặc gọi đến một API của bên thứ ba. Nếu thực hiện các công việc này một cách đồng bộ, người dùng sẽ phải chờ đợi rất lâu, gây ra trải nghiệm kém.

Để giải quyết vấn đề này, dự án đã triển khai một hệ thống hàng đợi tác vụ bất đồng bộ bằng cách kết hợp hai công nghệ là Celery và Redis.

* Celery - Hệ thống Hàng đợi Tác vụ (Task Queue):
  + Celery là một hệ thống hàng đợi tác vụ phân tán, mã nguồn mở, được viết bằng Python. Vai trò chính của nó là nhận các "công việc" (tasks) từ ứng dụng Django, đưa chúng vào một hàng đợi, và để các tiến trình riêng biệt (gọi là "workers") lấy và thực thi các công việc đó ở chế độ nền (background).
  + Công dụng: Celery giúp tách biệt các tác vụ tốn thời gian ra khỏi luồng xử lý chính. Điều này cho phép ứng dụng Django có thể phản hồi lại cho người dùng gần như ngay lập tức, trong khi công việc nặng vẫn đang được xử lý ở phía sau. Nó cũng hỗ trợ việc lên lịch cho các tác vụ chạy định kỳ (ví dụ: chạy vào lúc nửa đêm mỗi ngày).
* Redis - Kho lưu trữ Dữ liệu Trong bộ nhớ (In-memory Data Store):
  + Redis là một kho lưu trữ cấu trúc dữ liệu trong bộ nhớ, mã nguồn mở, được sử dụng như một cơ sở dữ liệu, bộ nhớ đệm (cache), và trình môi giới thông điệp (message broker). Nó nổi tiếng với tốc độ truy xuất cực kỳ nhanh do hoạt động chủ yếu trên RAM.
  + Công dụng: Trong kiến trúc của dự án, Redis đóng hai vai trò quan trọng:
    1. Trình môi giới cho Celery (Message Broker): Redis hoạt động như một "bảng tin" trung gian. Khi Django muốn thực thi một tác vụ nền, nó sẽ "đăng" một thông điệp về tác vụ đó lên Redis. Các Celery worker sẽ liên tục theo dõi và lấy các thông điệp này để thực thi.
    2. Bộ nhớ đệm cho Django (Cache Backend): Redis được sử dụng để lưu trữ tạm thời các dữ liệu ít thay đổi hoặc có chi phí truy vấn cao, giúp tăng tốc độ phản hồi của API và giảm tải cho cơ sở dữ liệu PostgreSQL chính.

#### Cách triển khai

Việc tích hợp Celery và Redis vào dự án Django là một quá trình gồm nhiều bước, từ cấu hình, định nghĩa tác vụ, cho đến việc kích hoạt và quản lý các tiến trình worker. Quá trình này biến ứng dụng Django từ một hệ thống chỉ xử lý đồng bộ thành một hệ thống có khả năng thực thi các tác vụ bất đồng bộ một cách hiệu quả.

***Tổng quan các bước triển khai:***

1. Cài đặt và Cấu hình Môi trường: Cài đặt các thư viện cần thiết và thiết lập kết nối đến Redis server.
2. Khởi tạo Ứng dụng Celery: Tạo một instance Celery riêng cho dự án Django.
3. Định nghĩa các Tác vụ (Tasks): Viết các hàm Python và đánh dấu chúng là các tác vụ có thể thực thi bởi Celery.
4. Kích hoạt Tác vụ từ Mã nguồn Django: Gọi các tác vụ nền từ các views hoặc services khi có một sự kiện nghiệp vụ xảy ra.
5. Chạy các Tiến trình Worker: Khởi động các tiến trình Celery worker để chúng bắt đầu lắng nghe và xử lý các tác vụ từ hàng đợi.

***Chi tiết triển khai trong Landify:***

* Bước 1: Cài đặt và Cấu hình Môi trường
  + Cài đặt Thư viện: Các thư viện cần thiết được cài đặt vào môi trường ảo của dự án bằng pip:

pip install celery redis django-celery-beat

celery là thư viện chính, redis là client để Python giao tiếp với Redis, và django-celery-beat là một extension để lên lịch cho các tác vụ định kỳ.

* + Cấu hình Kết nối: Trong file landifyapis/settings.py, các thông tin kết nối đến Redis server (host, port, password) được đọc từ file .env. Các biến này sau đó được sử dụng để cấu hình hai mục quan trọng:
    - CELERY\_BROKER\_URL: Chỉ định Redis là "trình môi giới" (message broker), nơi Django sẽ gửi các thông điệp tác vụ đến.
    - CELERY\_RESULT\_BACKEND: Cũng chỉ định Redis là nơi lưu trữ kết quả của các tác vụ (nếu cần).

REDIS\_HOST = os.getenv('REDIS\_HOST')

CELERY\_BROKER\_URL = f"redis://:{REDIS\_PASSWORD}@{REDIS\_HOST}:{REDIS\_PORT}"

CELERY\_RESULT\_BACKEND = f"redis://:{REDIS\_PASSWORD}@{REDIS\_HOST}:{REDIS\_PORT}/0"

* **Bước 2: Khởi tạo Ứng dụng Celery**
  + Một file mới là landifyapis/celery.py được tạo ra. File này chịu trách nhiệm khởi tạo một instance của ứng dụng Celery.
  + Đầu tiên, nó thiết lập biến môi trường DJANGO\_SETTINGS\_MODULE để Celery biết cách tìm và nạp các cấu hình từ dự án Django.
  + Sau đó, một instance Celery được tạo ra và được cấu hình để đọc tất cả các thiết lập có tiền tố CELERY\_ từ file settings.py.
  + Cuối cùng, lệnh app.autodiscover\_tasks() được gọi. Lệnh này chỉ thị cho Celery tự động tìm kiếm tất cả các file có tên tasks.py bên trong các "app" đã được đăng ký của Django để nhận diện các tác vụ.
* **Bước 3: Định nghĩa các Tác vụ (Tasks)**
  + Trong các app có chứa logic cần xử lý nền, một file tasks.py được tạo ra. Bên trong file này, các hàm Python thông thường được viết và được "đánh dấu" là một tác vụ Celery bằng decorator @shared\_task.
  + **Ví dụ về tác vụ kiểm tra spam trong apps/common/tasks/listings.py:**

# apps/common/tasks/listings.py (đoạn mã minh họa)

from celery import shared\_task

from apps.listings.models import Listing

@shared\_task(name="listing.check\_for\_spam")

def check\_listing\_for\_spam(listing\_id: int):

try:

listing = Listing.objects.get(id=listing\_id)

# ... logic kiểm tra spam ...

if is\_spam\_detected:

listing.active = False

listing.save()

except Listing.DoesNotExist:

# ... xử lý lỗi ...

* + **Ví dụ về tác vụ định kỳ trong apps/common/tasks/stats.py:** Tương tự, hàm update\_total\_listings\_count được định nghĩa để đếm và cập nhật thống kê. Tác vụ này sau đó được lên lịch để chạy hàng ngày thông qua cấu hình CELERY\_BEAT\_SCHEDULE trong settings.py.
* **Bước 4: Kích hoạt Tác vụ từ Mã nguồn Django**
  + Tại những nơi trong mã nguồn Django cần thực thi một công việc nền, thay vì gọi trực tiếp hàm, phương thức .delay() hoặc .apply\_async() của tác vụ sẽ được gọi.
  + Ví dụ, trong file apps/listings/services.py, sau khi hàm create\_full\_listing đã tạo thành công một tin đăng trong CSDL, nó sẽ kích hoạt tác vụ kiểm tra spam bằng lệnh:

# apps/listings/services.py (đoạn mã minh họa)

from apps.common.tasks import listings as listing\_tasks

def create\_full\_listing(...):

# ... logic tạo tin đăng ...

listing = Listing.objects.create(...)

# Kích hoạt tác vụ nền thay vì xử lý đồng bộ

listing\_tasks.check\_listing\_for\_spam.delay(listing.id)

return listing

* + Khi lệnh .delay() được gọi, Celery sẽ đóng gói tên của tác vụ (listing.check\_for\_spam) và các tham số của nó (listing.id) thành một thông điệp và gửi nó vào hàng đợi trên Redis.
* Bước 5: Chạy các Tiến trình Worker và Beat
  + Để hệ thống có thể xử lý các tác vụ, các tiến trình Celery phải được khởi động và chạy song song với ứng dụng Django. Việc này được thực hiện thông qua các lệnh trong terminal:
    - celery -A landifyapis worker -l info: Lệnh này khởi động các "worker". Các worker sẽ kết nối đến Redis, liên tục theo dõi hàng đợi và thực thi bất kỳ tác vụ nào xuất hiện.
    - celery -A landifyapis beat -l info --scheduler django\_celery\_beat.schedulers:DatabaseScheduler: Lệnh này khởi động "beat", là một tiến trình đặc biệt chịu trách nhiệm lên lịch. Nó sẽ kiểm tra cấu hình CELERY\_BEAT\_SCHEDULE và gửi các tác vụ định kỳ vào hàng đợi đúng thời điểm đã định.

Bằng cách triển khai hệ thống này, Landify đã chuyển đổi thành công các tác vụ tốn thời gian thành các hoạt động bất đồng bộ, giúp tối ưu hóa đáng kể thời gian phản hồi của API và cải thiện trải nghiệm chung của người dùng.

### Django Channels

Django Channels là một thư viện mở rộng chính thức của Django, được thiết kế để bổ sung khả năng xử lý các giao thức hoạt động dựa trên kết nối lâu dài (long-running connections) vào Django, vượt ra ngoài khuôn khổ yêu cầu-phản hồi (request-response) ngắn hạn của giao thức HTTP truyền thống.

* Django Channels - Mở rộng Khả năng Xử lý của Django:
  + Về bản chất, Channels mở rộng Django để có thể xử lý không chỉ HTTP mà còn các giao thức khác như WebSocket. WebSocket là một giao thức truyền thông hai chiều, cho phép server và client có thể gửi dữ liệu cho nhau bất cứ lúc nào trên một kết nối duy nhất được duy trì liên tục. Điều này là nền tảng cho việc xây dựng các tính năng thời gian thực (real-time).
  + Công dụng: Trong dự án Landify, Django Channels là công nghệ cốt lõi để hiện thực hóa các tính năng yêu cầu sự tương tác tức thời, mà quan trọng nhất là hệ thống Chat. Nó cho phép server có thể "đẩy" (push) tin nhắn mới đến ứng dụng của người nhận ngay khi nó được gửi đi, thay vì người nhận phải liên tục gửi yêu cầu hỏi "có tin nhắn mới không?".
* Redis - Lớp Kênh (Channel Layer):
  + Để các instance Django khác nhau (trong môi trường triển khai thực tế) có thể giao tiếp với nhau, Channels sử dụng một khái niệm gọi là "Channel Layer". Đây là một lớp trừu tượng hoạt động như một hệ thống "bưu điện", cho phép các phần khác nhau của ứng dụng gửi và nhận thông điệp.
  + Công dụng: Trong Landify, Redis được sử dụng làm backend cho Channel Layer. Khi một người dùng gửi tin nhắn, ChatConsumer sẽ không gửi trực tiếp đến người nhận. Thay vào đó, nó sẽ gửi tin nhắn đó vào một "kênh" (channel) trên Redis. Tất cả các ChatConsumer khác đang lắng nghe kênh này (tức là những người dùng khác trong cùng phòng chat) sẽ nhận được thông điệp và sau đó đẩy dữ liệu xuống cho client của họ qua kết nối WebSocket.

#### Cách triển khai Django Channels trong dự án Landify

Việc tích hợp Django Channels để xây dựng tính năng Chat real-time là một quá trình đòi hỏi sự cấu hình ở nhiều cấp độ, từ việc thay đổi cách ứng dụng khởi động cho đến việc viết logic xử lý kết nối WebSocket.

**Tổng quan các bước triển khai:**

1. **Cài đặt Thư viện và Cấu hình ASGI:** Cài đặt các gói cần thiết và chuyển đổi dự án từ chế độ WSGI (mặc định cho HTTP) sang ASGI (Asynchronous Server Gateway Interface) để có thể xử lý đồng thời cả HTTP và WebSocket.
2. **Thiết lập Lớp Kênh (Channel Layer):** Cấu hình Redis làm backend cho Channel Layer, cho phép các instance khác nhau của ứng dụng có thể giao tiếp với nhau.
3. **Định nghĩa Logic Xử lý WebSocket (Consumers):** Viết các lớp Consumer để xử lý các sự kiện của một kết nối WebSocket, chẳng hạn như khi một client kết nối, ngắt kết nối, hoặc gửi dữ liệu.
4. **Định tuyến Kết nối (Routing):** Tạo một hệ thống định tuyến riêng cho các kết nối WebSocket, chỉ định Consumer nào sẽ xử lý các kết nối đến một URL cụ thể.
5. **Tích hợp Xác thực:** Xây dựng một middleware tùy chỉnh để xác thực người dùng (trong trường hợp này là qua Firebase ID Token) ngay khi kết nối WebSocket được thiết lập.

**Chi tiết triển khai trong Landify:**

* **Bước 1: Cài đặt và Cấu hình ASGI**
  + Cài đặt Thư viện: Các thư viện cần thiết được cài đặt bằng pip:

pip install channels channels-redis daphne

channels là thư viện chính, channels-redis cung cấp backend Channel Layer sử dụng Redis, và daphne là một ASGI server được khuyến nghị để chạy các ứng dụng Channels trong môi trường production.

* + Đăng ký App: channels được thêm vào danh sách INSTALLED\_APPS trong settings.py.
  + Chuyển sang ASGI: Một file mới là landifyapis/asgi.py được tạo ra. File này sẽ thay thế wsgi.py làm điểm khởi động chính của ứng dụng. Biến ASGI\_APPLICATION trong settings.py được thiết lập để trỏ đến file này:

# landifyapis/settings.py

ASGI\_APPLICATION = "landifyapis.asgi.application"

* **Bước 2: Thiết lập Lớp Kênh (Channel Layer)**
  + Trong settings.py, mục CHANNEL\_LAYERS được cấu hình để sử dụng Redis. Nó chỉ định channels\_redis.core.RedisChannelLayer làm backend và cung cấp thông tin kết nối đến Redis server.

# landifyapis/settings.py (đoạn mã minh họa)

CHANNEL\_LAYERS = {

"default": {

"BACKEND": "channels\_redis.core.RedisChannelLayer",

"CONFIG": {

"hosts": [f"redis://:{REDIS\_PASSWORD}@{REDIS\_HOST}:{REDIS\_PORT}/0"],

},

},

}

* **Bước 3: Định nghĩa Logic Xử lý WebSocket (Consumers)**
  + Trong app apps.interactions, một file consumers.py được tạo ra để chứa logic cho tính năng Chat.
  + Lớp ChatConsumer được tạo bằng cách kế thừa từ AsyncWebsocketConsumer của Channels. Lớp này định nghĩa các phương thức async để xử lý các sự kiện chính:
    - connect(): Được gọi khi một client cố gắng thiết lập kết nối. Tại đây, mã nguồn sẽ trích xuất chat\_id từ URL, xác thực quyền truy cập của người dùng, và thêm kết nối này vào một "group" tương ứng với phòng chat.
    - disconnect(): Được gọi khi kết nối bị đóng. Mã nguồn sẽ xóa kết nối khỏi group.
    - receive(): Được gọi mỗi khi client gửi một thông điệp qua WebSocket. Tại đây, mã nguồn sẽ phân tích dữ liệu JSON, lưu tin nhắn vào CSDL PostgreSQL, và sau đó gửi lại tin nhắn đó đến toàn bộ group thông qua Channel Layer.
    - chat\_message(): Một hàm tùy chỉnh, được gọi khi ChatConsumer nhận được một thông điệp từ Channel Layer. Hàm này sẽ đóng gói dữ liệu và gửi nó xuống cho client qua kết nối WebSocket.
* **Bước 4: Định tuyến Kết nối (Routing)**
  + Tương tự như urls.py cho HTTP, một file apps/interactions/routing.py được tạo để xử lý các kết nối WebSocket. File này định nghĩa một websocket\_urlpatterns, sử dụng re\_path để khớp các URL của WebSocket (ví dụ: api/ws/chat/<chat\_id>/) với ChatConsumer tương ứng.
  + File landifyapis/asgi.py sau đó được cập nhật để sử dụng ProtocolTypeRouter. Router này hoạt động như một bộ chuyển mạch chính: nếu là một request HTTP, nó sẽ chuyển cho ứng dụng Django WSGI/ASGI tiêu chuẩn; nếu là một kết nối WebSocket (websocket), nó sẽ chuyển cho hệ thống định tuyến của Channels (URLRouter).

# landifyapis/asgi.py (đoạn mã minh họa)

application = ProtocolTypeRouter({

"http": get\_asgi\_application(),

"websocket": FirebaseTokenAuthMiddleware(

URLRouter(

apps.interactions.routing.websocket\_urlpatterns

)

),

})

* **Bước 5: Tích hợp Xác thực**
  + Xác thực cho WebSocket phức tạp hơn HTTP vì nó không có header cho mỗi thông điệp. Do đó, việc xác thực phải được thực hiện một lần duy nhất khi kết nối được thiết lập.
  + Trong Landify, token xác thực của Firebase được client gửi qua query string của URL WebSocket (ví dụ: ...?token=...).
  + Một middleware tùy chỉnh là FirebaseTokenAuthMiddleware (landifyapis/firebase\_auth\_middleware.py) được tạo ra. Middleware này sẽ bao bọc URLRouter của Channels. Trước khi chuyển kết nối đến ChatConsumer, nó sẽ trích xuất token từ query string, xác minh nó bằng Firebase Admin SDK, và gán đối tượng user đã được xác thực vào scope (tương đương với request trong HTTP). ChatConsumer sau đó có thể truy cập self.scope['user'] để biết người dùng nào đang kết nối.

Thông qua các bước cấu hình và lập trình này, Django Channels đã được tích hợp thành công vào Landify, mở rộng khả năng của hệ thống để hỗ trợ các tương tác hai chiều và thời gian thực, tạo ra một trải nghiệm Chat mượt mà và tức thời cho người dùng.

### Quản lý Trạng thái Nâng cao trong Flutter với BLoC và MVVM

#### Giới thiệu

Trong một ứng dụng đơn giản, việc quản lý trạng thái (state management) – tức là quản lý dữ liệu và cách nó thay đổi theo thời gian để cập nhật giao diện người dùng – có thể được xử lý bằng các cơ chế tích hợp sẵn của Flutter như setState. Tuy nhiên, đối với một ứng dụng có quy mô lớn và nghiệp vụ phức tạp như Landify, việc chỉ dựa vào setState sẽ nhanh chóng dẫn đến một cấu trúc mã nguồn hỗn loạn, khó kiểm thử và khó bảo trì.

Để giải quyết thách thức này, dự án Landify đã áp dụng một kiến trúc quản lý trạng thái nâng cao, kết hợp hai mẫu thiết kế mạnh mẽ là BLoC (Business Logic Component) và MVVM (Model-View-ViewModel).

* BLoC (Business Logic Component) - Tách biệt Logic Nghiệp vụ:
  + BLoC là một mẫu thiết kế kiến trúc được cộng đồng Flutter ưa chuộng, với mục tiêu chính là tách biệt hoàn toàn logic nghiệp vụ (business logic) ra khỏi tầng giao diện người dùng (presentation layer). Thay vì để các widget tự mình gọi API hay xử lý dữ liệu, BLoC hoạt động như một lớp trung gian. Nó nhận các "sự kiện" (Events) từ giao diện, xử lý các sự kiện đó, và phát ra các "trạng thái" (States) mới. Giao diện sau đó chỉ cần lắng nghe và tự động cập nhật lại dựa trên các trạng thái này.
  + Công dụng: Trong Landify, BLoC là "bộ não" đằng sau mỗi tính năng. Ví dụ, HomepageBloc chịu trách nhiệm gọi API để tải danh sách tin đăng, ChatBloc quản lý việc gửi và nhận tin nhắn real-time, trong khi VerificationBloc điều phối quy trình eKYC nhiều bước. Việc tách biệt này giúp cho mã nguồn của giao diện trở nên cực kỳ đơn giản, chỉ tập trung vào việc "vẽ" và gửi sự kiện.
* MVVM (Model-View-ViewModel) - Tách biệt Logic Trình bày:
  + MVVM là một mẫu thiết kế kiến trúc tập trung vào việc phân tách tầng giao diện thành các thành phần nhỏ hơn nữa. Nó giới thiệu một khái niệm gọi là ViewModel, một lớp chuyên chịu trách nhiệm "chế biến" và chuẩn bị dữ liệu để giao diện có thể hiển thị một cách trực tiếp nhất.
  + Công dụng: Nếu BLoC giúp tách logic *nghiệp vụ*, thì MVVM giúp tách logic *trình bày*. Trong Landify, điều này được thể hiện qua hai loại model:
    1. Data Model (core/models): Các lớp này (ví dụ: ListingPreviewModel) chỉ đại diện cho dữ liệu thô, có cấu trúc y hệt như JSON trả về từ API.
    2. ViewModel (features/.../view\_models): Các lớp này (ví dụ: ListingPreviewViewModel) nhận vào một Data Model và chuyển đổi nó thành dữ liệu đã được định dạng sẵn cho UI. Ví dụ, nó sẽ chuyển một đối tượng DateTime thành chuỗi "Đăng hôm qua", hay định dạng một con số 1500000000 thành chuỗi "1.5 tỷ".

***Ưu điểm và Lý do áp dụng:***Sự kết hợp giữa BLoC và MVVM không phải là một sự lựa chọn ngẫu nhiên, mà là một chiến lược nhằm đạt được các mục tiêu sau:

* Khả năng Kiểm thử (Testability): Bằng cách tách biệt hoàn toàn logic ra khỏi giao diện, việc viết các bài kiểm thử đơn vị (unit tests) cho logic nghiệp vụ trong BLoC và logic trình bày trong ViewModel trở nên cực kỳ dễ dàng, giúp đảm bảo chất lượng và độ ổn định của ứng dụng.
* Khả năng Bảo trì và Mở rộng (Maintainability & Scalability): Khi một tính năng cần thay đổi, các nhà phát triển có thể xác định chính xác vị trí cần sửa đổi (trong BLoC, ViewModel, hay UI) mà không sợ ảnh hưởng đến các phần khác. Việc thêm tính năng mới cũng trở nên đơn giản hơn bằng cách tạo ra các BLoC và ViewModel mới.
* Luồng dữ liệu Rõ ràng và Dễ dự đoán: Kiến trúc BLoC áp đặt một luồng dữ liệu đơn hướng (UI → Event → BLoC → State → UI). Luồng dữ liệu này giúp cho việc theo dõi và gỡ lỗi (debug) các vấn đề trở nên dễ dàng hơn rất nhiều so với việc sử dụng setState một cách tự do, vốn có thể gây ra các thay đổi trạng thái khó lường.
* Tăng hiệu suất và Giảm Rebuild không cần thiết: Bằng cách sử dụng các thư viện như equatable và freezed để tạo ra các State bất biến, BLoC có thể so sánh hiệu quả trạng thái cũ và mới. BlocBuilder sẽ chỉ "vẽ lại" những phần giao diện thực sự cần thiết khi có sự thay đổi, giúp tối ưu hóa hiệu năng của ứng dụng.

#### Cách triển khai Kiến trúc BLoC và MVVM trong ứng dụng Landify

Việc hiện thực hóa kiến trúc BLoC kết hợp với MVVM trong Landify là một quy trình có hệ thống, được hỗ trợ bởi một bộ thư viện mạnh mẽ và các quy ước tổ chức mã nguồn chặt chẽ. Cách tiếp cận này đảm bảo rằng luồng dữ liệu trong ứng dụng luôn rõ ràng, dễ quản lý và có khả năng mở rộng.

***Tổng quan các bước triển khai:***

1. Cài đặt và Cấu hình các Thư viện Nền tảng: Tích hợp các gói thư viện cần thiết cho việc quản lý trạng thái và sinh mã.
2. Tổ chức Cấu trúc Thư mục theo Feature và Tầng: Phân chia mã nguồn thành các module chức năng, và trong mỗi module lại phân chia theo các tầng kiến trúc.
3. Định nghĩa Lớp Dữ liệu (Data Models): Tạo các lớp Dart để ánh xạ với dữ liệu JSON từ API.
4. Xây dựng Lớp Logic (BLoCs): Viết các lớp BLoC để xử lý logic nghiệp vụ, bao gồm việc định nghĩa các Events và States.
5. Xây dựng Lớp Trình bày (ViewModels): Tạo các lớp ViewModel để "chế biến" dữ liệu từ Data Models thành định dạng sẵn sàng cho UI.
6. Kết nối các Tầng trong Giao diện (UI): Sử dụng các widget như BlocProvider, BlocBuilder, và BlocListener để kết nối giao diện với BLoC và hiển thị dữ liệu từ ViewModel.

***Chi tiết triển khai trong Landify:***

* Bước 1: Cài đặt và Cấu hình Thư viện
  + Các thư viện cốt lõi cho việc quản lý trạng thái được thêm vào file pubspec.yaml:
    - flutter\_bloc: Cung cấp các widget và lớp cơ sở để triển khai kiến trúc BLoC.
    - freezed & freezed\_annotation: Được sử dụng để sinh mã cho các lớp State và Event, giúp tạo ra các đối tượng bất biến (immutable) và các phương thức tiện ích như copyWith, map, when.
    - equatable: Được sử dụng cho các lớp DataModel và ViewModel để đơn giản hóa việc so sánh các đối tượng dựa trên giá trị thuộc tính thay vì tham chiếu bộ nhớ.
* Bước 2: Tổ chức Cấu trúc Thư mục
  + Dự án tuân thủ một cấu trúc thư mục nghiêm ngặt. Bên trong lib/features/, mỗi tính năng (ví dụ: listing\_detail) được chia thành ba tầng chính:
    - data/: Chứa repositories/ (gọi API và trả về Data Models) và models/ (định nghĩa Data Models).
    - logic/: Chứa các file của BLoC (...\_bloc.dart, ...\_event.dart, ...\_state.dart).
    - presentation/: Chứa screens/ (các màn hình hoàn chỉnh), widgets/ (các thành phần giao diện con), và view\_models/ (định nghĩa ViewModels).
* Bước 3: Định nghĩa Lớp Dữ liệu (Data Models)
  + Trong thư mục core/models/, các lớp như ListingPreviewModel.dart được tạo ra. Chúng kế thừa từ Equatable và chỉ chứa các thuộc tính tương ứng với các key trong JSON trả về từ API, cùng với một factory constructor fromJson.

// core/models/listing/listing\_preview\_model.dart (đoạn mã minh họa)

class ListingPreviewModel extends Equatable {

final String publicId;

final String title;

final DateTime? createdDate;

// ... các trường khác

const ListingPreviewModel({...});

factory ListingPreviewModel.fromJson(Map<String, dynamic> json) {

return ListingPreviewModel(

publicId: json['public\_id'],

title: json['title'],

createdDate: DateTime.tryParse(json['created\_date']),

// ...

);

}

// ...

}

* Bước 4: Xây dựng Lớp Logic (BLoCs)
  + Đối với mỗi tính năng, một BLoC được tạo ra. Ví dụ, ListingDetailBloc (features/listing\_detail/logic/).
  + Events và States được định nghĩa trong các file riêng (...\_event.dart, ...\_state.dart) bằng cách sử dụng freezed. Điều này cho phép tạo ra các lớp sealed class, giúp cho việc xử lý các event và state khác nhau trong BLoC trở nên an toàn và rõ ràng hơn thông qua phương thức map hoặc when.
  + Logic xử lý: Bên trong ListingDetailBloc, một handler được đăng ký cho ListingDetailFetched event. Handler này sẽ gọi đến \_listingDetailRepository để lấy về ListingDetailModel (Data Model).
* Bước 5: Xây dựng Lớp Trình bày (ViewModels)
  + Sau khi BLoC nhận được Data Model, nó sẽ không phát ra State chứa **trực tiếp Data Model đó. Thay vào đó, nó sẽ tạo ra một ViewModel.**
  + Ví dụ, ListingDetailViewModel (features/listing\_detail/presentation/view\_models/) được tạo ra. Nó có một factory constructor fromModel nhận vào ListingDetailModel. Bên trong constructor này, toàn bộ logic "chế biến" dữ liệu được thực hiện.

// features/listing\_detail/presentation/view\_models/listing\_detail\_view\_model.dart (minh họa)

class ListingDetailViewModel extends Equatable {

final String title;

final String formattedPrice;

// ... các trường khác đã được định dạng

factory ListingDetailViewModel.fromModel(ListingDetailModel model) {

return ListingDetailViewModel(

title: model.title.toUpperCase(), // Logic trình bày: viết hoa tiêu đề

formattedPrice: formatPrice(model.priceValue, model.unitPriceCode), // Logic trình bày: định dạng giá

// ...

);

}

// ...

}

* Bước 6: Kết nối các Tầng trong Giao diện (UI)
  + Cung cấp BLoC: Tại màn hình chính của một feature (ví dụ: ListingDetailPage), BlocProvider được sử dụng để tạo và cung cấp instance của BLoC cho toàn bộ cây widget con của màn hình đó.
  + Xây dựng Giao diện: Các widget con sử dụng BlocBuilder để lắng nghe State từ BLoC. Khi một State mới được phát ra, BlocBuilder sẽ tự động rebuild lại giao diện.
  + Hiển thị Dữ liệu: Bên trong builder của BlocBuilder, widget sẽ truy cập vào ViewModel đã được chứa sẵn trong State để lấy dữ liệu đã được định dạng và hiển thị trực tiếp, mà không cần phải thực hiện bất kỳ logic định dạng nào.

// features/listing\_detail/presentation/screens/detail\_screen.dart (minh họa)

BlocBuilder<ListingDetailBloc, ListingDetailState>(

builder: (context, state) {

if (state.status == ListingDetailStatus.success) {

// Lấy ViewModel từ State

final viewModel = state.listingViewModel!;

// Truyền ViewModel xuống các widget con

return Column(

children: [

PropertyInfoSection(viewModel: viewModel.info),

SellerCard(viewModel: viewModel.seller),

],

);

}

// ...

},

)

* + Xử lý Hành động một lần: BlocListener được sử dụng để xử lý các "side-effect" như hiển thị SnackBar khi có lỗi, hoặc điều hướng (Navigator.push) khi một hành động thành công, mà không cần phải rebuild lại toàn bộ giao diện.

Bằng cách áp dụng một cách nhất quán quy trình này trên toàn bộ ứng dụng, Landify đã xây dựng được một kiến trúc frontend vững chắc, nơi mỗi tầng có một trách nhiệm duy nhất, giúp cho mã nguồn trở nên sạch sẽ, dễ hiểu và có khả năng thích ứng cao với các thay đổi trong tương lai

## Tích hợp Dịch vụ của Bên thứ ba

Để tăng tốc độ phát triển và tận dụng sức mạnh của các nền tảng công nghệ chuyên biệt, dự án Landify đã tích hợp một cách chiến lược các dịch vụ của bên thứ ba (Third-Party Services). Cách tiếp cận này cho phép hệ thống tập trung vào việc xây dựng logic nghiệp vụ cốt lõi, trong khi các tác vụ phức tạp như xác thực, lưu trữ media hay giao tiếp thời gian thực được ủy thác cho các nhà cung cấp hàng đầu, vốn đã được kiểm chứng về độ tin cậy và khả năng mở rộng.

### Hệ sinh thái Firebase

Firebase, một nền tảng phát triển ứng dụng của Google, được tích hợp vào Landify như một giải pháp Backend-as-a-Service (BaaS) để xử lý các nghiệp vụ cụ thể về định danh và dữ liệu thời gian thực.

Dịch vụ Firebase Authentication được sử dụng làm nền tảng xác thực chính của ứng dụng. Nó cung cấp một cơ chế an toàn và linh hoạt, hỗ trợ nhiều phương thức đăng nhập như số điện thoại (OTP) và các nhà cung cấp định danh liên kết như Google. Việc triển khai được thực hiện ở phía ứng dụng Flutter thông qua các thư viện firebase\_auth và google\_sign\_in. Khi người dùng đăng nhập thành công, Firebase sẽ cấp phát một ID Token (JWT). Token này sau đó được gửi đến backend Django, nơi thư viện Firebase Admin SDK sẽ xác minh tính hợp lệ và trích xuất firebase\_uid để ánh xạ với tài khoản tương ứng trong cơ sở dữ liệu PostgreSQL. Cách làm này giúp giảm tải cho backend chính khỏi việc phải quản lý và bảo mật thông tin đăng nhập nhạy cảm của người dùng.

Bên cạnh đó, Cloud Firestore, một cơ sở dữ liệu NoSQL, được tích hợp để giải quyết bài toán đồng bộ hóa dữ liệu theo thời gian thực. Trong Landify, nó được sử dụng chủ yếu cho tính năng Thông báo. Phía backend Django, khi có một sự kiện cần thông báo, nó sẽ sử dụng Firebase Admin SDK để ghi một document mới vào collection của người dùng trên Firestore. Phía ứng dụng Flutter, thư viện cloud\_firestore được dùng để "lắng nghe" (listen) các thay đổi trên collection này. Ngay khi có dữ liệu mới, Firestore sẽ tự động "đẩy" (push) xuống tất cả các client đang lắng nghe, giúp cập nhật giao diện một cách tức thời mà không cần người dùng phải thực hiện thao tác làm mới.

### Cloudinary

Cloudinary được lựa chọn làm nền tảng quản lý tài sản đa phương tiện (media assets) trên đám mây, cung cấp một giải pháp toàn diện cho việc tải lên, lưu trữ, tối ưu hóa và phân phối hình ảnh, video. Vai trò của Cloudinary là giải phóng backend chính khỏi gánh nặng lưu trữ và xử lý các tệp media có dung lượng lớn.

Việc tích hợp được thực hiện ở tầng backend Django thông qua các thư viện cloudinary và django-cloudinary-storage. Trong các model liên quan như UserProfile và PropertyMedia, các trường chứa đường dẫn tệp được định nghĩa là kiểu CloudinaryField. Khi người dùng tải lên một tệp hình ảnh, thay vì lưu trữ trên server, ứng dụng Django sẽ tự động sử dụng Cloudinary SDK để tải tệp đó lên dịch vụ Cloudinary. Dịch vụ này sẽ xử lý, tối ưu hóa (nén, thay đổi kích thước nếu cần) và trả về một URL an toàn. Cuối cùng, cơ sở dữ liệu PostgreSQL chỉ cần lưu lại đường dẫn URL này. Khi hiển thị, ứng dụng Flutter sẽ tải ảnh trực tiếp từ URL của Cloudinary, tận dụng mạng lưới phân phối nội dung (CDN) toàn cầu để đảm bảo tốc độ tải nhanh nhất cho người dùng cuối.

### Agora

Để hiện thực hóa tính năng gọi video chất lượng cao, dự án đã tích hợp Agora, một nền tảng cung cấp API và SDK chuyên dụng cho các ứng dụng giao tiếp thời gian thực (Real-Time Engagement - RTE). Agora cung cấp một hạ tầng mạng lưới toàn cầu, được tối ưu hóa để truyền tải dữ liệu âm thanh và video với độ trễ thấp, một bài toán kỹ thuật phức tạp mà việc tự xây dựng sẽ rất tốn kém.

Luồng triển khai của tính năng này được phân chia rõ ràng giữa backend và frontend. Khi một người dùng muốn bắt đầu cuộc gọi, ứng dụng Flutter sẽ gửi yêu cầu đến một endpoint API trên Django. Phía backend, sử dụng thư viện agora\_token\_builder, sẽ tạo ra một token truy cập tạm thời và an toàn, chứa thông tin về quyền hạn của người dùng trong kênh thoại đó. Token này sau đó được trả về cho ứng dụng Flutter. Phía client, sử dụng Agora RTC SDK (agora\_rtc\_engine), sẽ dùng token này để kết nối trực tiếp đến các máy chủ của Agora và tham gia vào kênh. Sau khi kết nối được thiết lập, toàn bộ việc truyền tải dữ liệu âm thanh và video sẽ được thực hiện trực tiếp giữa các client thông qua hạ tầng của Agora, đảm bảo hiệu năng và chất lượng cuộc gọi tối ưu.

# HỆ THỐNG GIAO LƯU MUA BÁN BẤT ĐỘNG SẢN

*Chương này trình bày chi tiết về hệ thống giao dịch bất động sản Landify được xây dựng trong khuôn khổ đồ án. Nội dung chương sẽ tập trung vào việc giới thiệu tổng quan đề tài, sau đó đi sâu vào phân tích kiến trúc hệ thống, diễn giải các thành phần công nghệ và cơ chế hoạt động phối hợp giữa chúng.*

## Giới thiệu đề tài

Hệ thống Landify được phát triển với mục tiêu xây dựng một nền tảng bất động sản toàn diện, hiện đại và đáng tin cậy, giải quyết các nhu cầu đa dạng của cả người mua, người bán, và nhà môi giới. Đây không chỉ là một kênh đăng tin đơn thuần, mà còn là một hệ sinh thái tích hợp đầy đủ các công cụ và tính năng cần thiết để nâng cao trải nghiệm người dùng và tối ưu hóa quy trình giao dịch.

* Về phía người dùng, Landify mang đến một trải nghiệm mượt mà và trực quan thông qua ứng dụng di động (xây dựng bằng Flutter). Người dùng có thể dễ dàng truy cập vào một kho tin đăng phong phú, mỗi tin đăng đều được cung cấp thông tin chi tiết, đa dạng từ hình ảnh, video, thông số kỹ thuật, cho đến vị trí chính xác trên bản đồ. Hệ thống tìm kiếm và bộ lọc thông minh cho phép người dùng nhanh chóng tìm thấy bất động sản phù hợp với nhu cầu. Đặc biệt, để xây dựng một môi trường giao dịch an toàn và minh bạch, dự án đã tích hợp các tính năng cao cấp như:
  + Xác thực danh tính (eKYC): Tăng cường sự tin cậy giữa các bên tham gia.
  + Tương tác thời gian thực: Cho phép người dùng Chat, Đặt lịch hẹn và cả Gọi video trực tiếp trên ứng dụng.
  + Xây dựng cộng đồng: Tích hợp các tính năng mạng xã hội như bài đăng, bình luận và đánh giá, giúp người dùng chia sẻ kinh nghiệm và tham khảo ý kiến.
* Về phía quản trị, Landify cung cấp một hệ thống quản lý tập trung và mạnh mẽ (xây dựng trên Django Admin). Đội ngũ quản trị viên có toàn quyền kiểm soát các hoạt động trên nền tảng:
  + Quản lý nội dung: Duyệt, chỉnh sửa, hoặc từ chối các tin đăng không phù hợp.
  + Quản lý người dùng: Theo dõi và quản lý thông tin tài khoản người dùng.
  + Hệ thống kiểm duyệt: Xử lý các báo cáo vi phạm từ người dùng và giải quyết các kháng nghị một cách có hệ thống.
  + Thống kê và báo cáo: Theo dõi các chỉ số quan trọng về hoạt động của hệ thống để đưa ra các quyết định vận hành phù hợp.

## Khảo sát và chiến lược xây dựng hệ thống Landify

### Giới thiệu và Mục tiêu Khảo sát

Trước khi bắt tay vào việc thiết kế và xây dựng hệ thống Landify, việc tiến hành một cuộc khảo sát sâu rộng về các nền tảng bất động sản hiện có là một bước đi mang tính chiến lược. Thị trường công nghệ bất động sản (PropTech) tại Việt Nam là một môi trường năng động nhưng cũng đầy cạnh tranh, với sự hiện diện của cả những "ông lớn" lâu năm và các startup mới nổi. Việc chỉ xây dựng một sản phẩm mà không hiểu rõ bối cảnh thị trường, các tiêu chuẩn ngành, và quan trọng nhất là những "nỗi đau" (pain points) mà người dùng cuối đang thực sự gặp phải, sẽ dẫn đến nguy cơ tạo ra một sản phẩm không có lợi thế cạnh tranh và không đáp ứng được nhu cầu thực tế.

Do đó, quá trình khảo sát này không chỉ dừng lại ở việc liệt kê tính năng của đối thủ, mà còn là một hoạt động nghiên cứu toàn diện nhằm phân tích mô hình hoạt động, trải nghiệm người dùng, và các chiến lược công nghệ mà các nền tảng khác đang áp dụng. Kết quả từ cuộc khảo sát này chính là kim chỉ nam, là nền tảng vững chắc để định hình tầm nhìn, xác định các tính năng độc đáo và đưa ra những quyết định kiến trúc quan trọng cho hệ thống Landify.

***Mục tiêu chính*** của quá trình khảo sát được xác định rõ ràng như sau:

1. *Xác định các tính năng cốt lõi và tiêu chuẩn ngành:* Phân tích các nền tảng hàng đầu để nhận diện bộ tính năng tối thiểu (minimum viable product) mà bất kỳ ứng dụng bất động sản nào cũng phải có, từ đó đảm bảo Landify đáp ứng được kỳ vọng cơ bản của người dùng. Các tính năng này bao gồm quy trình đăng tin, bộ lọc tìm kiếm chi tiết, quản lý tài khoản, và các kênh liên lạc cơ bản.
2. *Phân tích ưu điểm và nhược điểm của các đối thủ cạnh tranh:*
   * Học hỏi từ ưu điểm: Nghiên cứu những điểm mạnh của các đối thủ như Batdongsan.com.vn về độ phủ dữ liệu, hay Nhatot.com về sự đơn giản trong giao diện, để chắt lọc những kinh nghiệm thành công và áp dụng một cách phù hợp cho Landify.
   * Cải tiến từ nhược điểm: Tập trung tìm kiếm những vấn đề còn tồn tại trên các nền tảng khác, chẳng hạn như vấn đề "tin rác", quy trình xác thực người dùng lỏng lẻo, trải nghiệm tương tác rời rạc, hay giao diện người dùng phức tạp. Đây chính là những "khoảng trống" thị trường, là cơ hội để Landify tạo ra sự khác biệt.
3. *Tìm kiếm cơ hội đổi mới và tạo lợi thế cạnh tranh:* Vượt ra ngoài các tính năng cơ bản, mục tiêu của khảo sát là tìm kiếm những hướng đi mới mà các đối thủ chưa khai thác hoặc làm chưa tốt. Điều này bao gồm việc phân tích khả năng ứng dụng các công nghệ mới như Trí tuệ nhân tạo (AI) để cá nhân hóa trải nghiệm (ví dụ: tư vấn phong thủy), tăng cường bảo mật (eKYC), và xây dựng các công cụ tương tác thời gian thực (Video Call, Interactive Chat) để giải quyết các vấn đề cốt lõi trong quá trình giao dịch.
4. *Làm cơ sở cho các quyết định về kiến trúc công nghệ:* Việc phân tích hiệu năng, tốc độ phản hồi và trải nghiệm người dùng trên các ứng dụng di động của đối thủ giúp củng cố quyết định lựa chọn một kiến trúc hiện đại cho Landify. Cụ thể, nhận thấy sự cần thiết của các tính năng real-time và hiệu năng mượt mà, nhóm đã quyết định lựa chọn Flutter (với kiến trúc BLoC + MVVM) cho frontend và một hệ thống backend mạnh mẽ, bất đồng bộ với Django, Celeryvà Django Channels.

Như vậy có thể thấy rằng: Quá trình khảo sát không chỉ là một bước thủ tục mà là giai đoạn nền tảng, giúp Landify được định hình không phải như một bản sao, mà là một phiên bản cải tiến, giải quyết những vấn đề thực tiễn và mang lại giá trị độc đáo cho người dùng trong hệ sinh thái bất động sản số.

### Phương pháp và Đối tượng Khảo sát

Để đảm bảo tính khách quan và toàn diện, quá trình khảo sát được thực hiện theo một phương pháp đa diện, không chỉ phân tích các tính năng bề mặt mà còn đi sâu vào mô hình hoạt động và trải nghiệm người dùng thực tế. Nhóm đã kết hợp hai phương pháp chính:

(1) **Phân tích trực tiếp sản phẩm**, tự mình trải nghiệm các luồng chức năng chính trên cả nền tảng website và ứng dụng di động của đối thủ.

(2) **Nghiên cứu thứ cấp**, tổng hợp các đánh giá, phản hồi của người dùng trên các kho ứng dụng (App Store, Google Play) và các diễn đàn công nghệ để có được một bức tranh tổng thể về những gì thị trường đang làm tốt và những gì còn thiếu sót.

Đối tượng khảo sát được lựa chọn một cách có chủ đích để bao quát các phân khúc khác nhau của thị trường PropTech tại Việt Nam, từ những nền tảng dẫn đầu cho đến các mô hình kinh doanh độc đáo.

1. ***Những người dẫn đầu thị trường (Market Leaders):***
   * *Batdongsan.com.vn:* Là nền tảng lớn nhất và lâu đời nhất, đây được xem là tiêu chuẩn của ngành. Việc khảo sát Batdongsan.com.vn giúp xác định bộ tính năng toàn diện nhất, từ hệ thống tìm kiếm, lọc chi tiết cho đến các công cụ dành cho nhà môi giới chuyên nghiệp. Đây là nguồn tham khảo quan trọng để xây dựng các chức năng cốt lõi cho Landify.
   * *Alonhadat.com.vn:* Một đối thủ mạnh khác, tập trung vào việc kết nối trực tiếp với các nhà môi giới truyền thống. Phân tích nền tảng này giúp hiểu rõ hơn về nhu cầu và thói quen làm việc của đối tượng người dùng là các nhà môi giới.
2. ***Nền tảng rao vặt đa ngành (Multi-category Classifieds):***
   * *Nhatot.com (mảng Bất động sản):* Đại diện cho một cách tiếp cận khác, tập trung vào sự đơn giản, nhanh chóng và hướng đến các giao dịch ngang hàng (peer-to-peer). Trải nghiệm người dùng trên Nhatot là bài học quý giá về việc làm thế nào để tối ưu hóa luồng đăng tin và tìm kiếm cho người dùng không chuyên.
3. ***Các nền tảng ứng dụng công nghệ mới (Technology-driven Platforms):***
   * *Guland.vn:* Nổi bật với việc lấy bản đồ làm trung tâm, tích hợp sâu dữ liệu quy hoạch. Việc khảo sát Guland củng cố cho quyết định của Landify trong việc đầu tư mạnh mẽ vào PostGIS và các tính năng hiển thị dữ liệu trên bản đồ một cách trực quan.
   * *Propzy (giai đoạn còn hoạt động mạnh):* Là một ví dụ điển hình về mô hình O2O (Online-to-Offline), kết hợp nền tảng công nghệ với đội ngũ tư vấn viên thực địa. Phân tích mô hình này giúp Landify định hình các tính năng hỗ trợ tương tác và đặt lịch hẹn một cách hiệu quả hơn.

**Tiêu chí đánh giá** được xây dựng một cách có hệ thống để đảm bảo mọi khía cạnh quan trọng của một nền tảng bất động sản đều được xem xét:

* *Trải nghiệm Người dùng (UI/UX):* Đánh giá sự mượt mà, logic và tính trực quan của các luồng chức năng chính: đăng ký/đăng nhập, đăng tin, tìm kiếm và lọc, xem chi tiết tin đăng. Mục tiêu là xác định các điểm gây khó khăn (friction points) cho người dùng để thiết kế một trải nghiệm tốt hơn cho Landify.
* *Hệ thống Tính năng Cốt lõi:* Phân tích chiều sâu và sự đa dạng của bộ lọc tìm kiếm, mức độ chi tiết của thông tin trong một tin đăng, khả năng quản lý media (hình ảnh, video), và các công cụ quản lý tài khoản cá nhân.
* *Tính Minh bạch và Tin cậy:* Đây là tiêu chí quan trọng nhất. Nhóm tập trung phân tích các cơ chế xác thực người dùng (nếu có), quy trình kiểm duyệt nội dung, cách các nền tảng xử lý "tin rác" (spam), và sự tồn tại của hệ thống đánh giá người bán/nhà môi giới. Sự yếu kém của các nền tảng hiện tại ở khía cạnh này chính là động lực lớn nhất để Landify xây dựng tính năng eKYC.
* *Công cụ Tương tác và Giao tiếp:* Khảo sát các công cụ có sẵn để người mua và người bán kết nối, bao gồm hệ thống chat tích hợp, nút gọi điện, và các tính năng đặt lịch hẹn. Hầu hết các hệ thống hiện tại chỉ cung cấp các công cụ cơ bản, mở ra cơ hội cho Landify phát triển một hệ sinh thái tương tác toàn diện hơn với Chat Real-time, Video Call, và các "Thẻ tương tác" nghiệp vụ.
* *Ứng dụng Công nghệ và Đổi mới:* Tìm kiếm các yếu tố công nghệ tạo nên sự khác biệt, chẳng hạn như việc tích hợp bản đồ GIS, phân tích dữ liệu thị trường, hoặc ứng dụng Trí tuệ nhân tạo. Việc nhận thấy sự thiếu vắng các công cụ tư vấn cá nhân hóa đã trực tiếp dẫn đến ý tưởng xây dựng microservice Chatbot tư vấn Phong thủy cho Landify.

Thông qua việc khảo sát có hệ thống với các đối tượng và tiêu chí rõ ràng, nhóm đã có được một nền tảng thông tin vững chắc. Các kết quả đúc kết từ quá trình này sẽ được trình bày chi tiết ở mục tiếp theo, làm cơ sở để giải thích cho các quyết định chiến lược trong việc xây dựng kiến trúc và tính năng của hệ thống Landify.

### Phân tích và Đúc kết Kết quả

Sau quá trình khảo sát và trải nghiệm thực tế các nền tảng hàng đầu, nhóm đã tổng hợp và phân tích các kết quả thu được. Quá trình này không chỉ giúp nhận diện những điểm mạnh cần học hỏi mà còn, quan trọng hơn, đã vạch ra những hạn chế cố hữu và "khoảng trống" còn tồn tại trên thị trường. Những đúc kết này chính là cơ sở để định hình nên chiến lược phát triển sản phẩm khác biệt và vượt trội cho Landify.

Các nền tảng bất động sản lớn tại Việt Nam đã đạt được những thành công nhất định và trở thành tiêu chuẩn mà bất kỳ sản phẩm mới nào cũng cần phải đáp ứng. Landify đã học hỏi và kế thừa những điểm mạnh sau:

* *Kho dữ liệu tin đăng khổng lồ:* Các nền tảng như Batdongsan.com.vn đã chứng minh rằng số lượng tin đăng phong phú là yếu tố then chốt để thu hút và giữ chân người dùng. Điều này khẳng định tầm quan trọng của việc xây dựng một hệ thống backend có khả năng mở rộng, có thể xử lý hàng triệu bản ghi một cách hiệu quả.
* *Hệ thống tìm kiếm và lọc chi tiết:* Các bộ lọc đa dạng theo vị trí, mức giá, diện tích, số phòng ngủ, hướng nhà... là tính năng bắt buộc. Landify đã kế thừa và phát triển một hệ thống lọc mạnh mẽ hơn nữa bằng cách sử dụng PostGIS cho các truy vấn địa lý nâng cao và một cấu trúc PropertyFeature linh hoạt.
* *Mô hình kinh doanh đã được kiểm chứng:* Mô hình doanh thu dựa trên việc thu phí cho các gói tin VIP, dịch vụ đẩy tin, và quảng cáo banner đã được thị trường chấp nhận. Landify đã xây dựng sẵn các model VipType và PromotionRule để có thể triển khai các mô hình kinh doanh tương tự một cách dễ dàng.
* *Giao diện trực quan, lấy bản đồ làm trung tâm:* Các nền tảng như Guland.vn cho thấy xu hướng người dùng ngày càng muốn trực quan hóa thông tin trên bản đồ. Điều này củng cố quyết định của Landify trong việc tích hợp sâu bản đồ vào cả luồng tìm kiếm và hiển thị chi tiết bất động sản.

Đây là những vấn đề cốt lõi mà Landify được sinh ra để giải quyết. Quá trình khảo sát đã chỉ ra bốn "khoảng trống" lớn nhất trên thị trường hiện tại:

1. *Khủng hoảng Niềm tin và Thiếu Minh bạch:*
   * Vấn đề: Đây là "nỗi đau" lớn nhất của người dùng. Tình trạng "tin rác", tin đăng ảo với giá không thực tế, thông tin pháp lý mập mờ, và người đăng không rõ danh tính vẫn còn tràn lan. Hầu hết các nền tảng chỉ dừng lại ở việc xác thực số điện thoại, một biện pháp bảo mật còn khá lỏng lẻo và chưa đủ để xây dựng lòng tin.
   * Cơ hội cho Landify: Đây là cơ hội lớn nhất. Landify quyết định đầu tư vào một giải pháp triệt để hơn: xây dựng quy trình xác thực danh tính điện tử (eKYC) toàn diện. Bằng cách yêu cầu người dùng (đặc biệt là người đăng tin) xác thực CCCD và khuôn mặt, Landify hướng đến việc tạo ra một cộng đồng người dùng có danh tính rõ ràng, từ đó giảm thiểu rủi ro lừa đảo và nâng cao chất lượng tin đăng một cách tự nhiên.
2. *Trải nghiệm Tương tác Rời rạc và Thiếu Hiệu quả:*
   * Vấn đề: Các công cụ giao tiếp trên các nền tảng hiện tại thường rất cơ bản. Hệ thống chat (nếu có) thường chậm, thiếu các tính năng nâng cao và không được tích hợp sâu vào quy trình giao dịch. Việc đặt lịch hẹn hay thương lượng thường phải chuyển qua các kênh bên ngoài (Zalo, gọi điện), làm đứt gãy luồng trải nghiệm và khó khăn cho việc theo dõi.
   * Cơ hội cho Landify: Landify không xem chat là một tính năng phụ, mà là trung tâm của sự tương tác. Bằng cách kết hợp Django Channels(WebSocket) và Cloud Firestore, hệ thống chat của Landify đảm bảo tốc độ real-time. Quan trọng hơn, nó được làm giàu bằng các "Thẻ tương tác" (Interactive Cards), cho phép người dùng thực hiện các nghiệp vụ như Đặt lịch hẹn hay Gửi yêu cầu hợp tác môi giới ngay trong giao diện trò chuyện. Việc tích hợp thêm Video Call (Agora) càng củng cố thêm vị thế của Landify như một nền tảng giao tiếp "tất cả trong một".
3. *Thiếu các Công cụ Hỗ trợ Ra quyết định Thông minh:*
   * Vấn đề: Hầu hết các nền tảng chỉ đóng vai trò là nơi hiển thị thông tin. Chúng cung cấp "dữ liệu" nhưng lại thiếu các công cụ để biến dữ liệu đó thành "thông tin chi tiết" (insight) hữu ích cho người dùng. Người mua vẫn phải tự mình tìm hiểu và phân tích các yếu tố phức tạp như quy hoạch, pháp lý, hay thậm chí là phong thủy.
   * Cơ hội cho Landify: Landify định vị mình không chỉ là một sàn giao dịch, mà còn là một "trợ lý ảo". Việc xây dựng microservice Chatbot tư vấn Phong thủy là bước đi đầu tiên trong chiến lược này. Bằng cách cung cấp một công cụ phân tích cá nhân hóa dựa trên năm sinh và hướng nhà—một yếu tố văn hóa quan trọng đối với người Việt—Landify đã tạo ra một giá trị gia tăng độc đáo, giúp người dùng tự tin hơn khi ra quyết định.
4. *Kiến trúc Công nghệ chưa Tối ưu cho Di động và Real-time:*
   * Vấn đề: Một số nền tảng lớn được xây dựng từ lâu với kiến trúc thiên về web truyền thống. Điều này đôi khi dẫn đến việc ứng dụng di động của họ chỉ là một "bản sao" của website, chưa tận dụng hết sức mạnh của nền tảng di động và có thể gặp vấn đề về hiệu năng cũng như trải nghiệm người dùng.
   * Cơ hội cho Landify: Landify được xây dựng với triết lý "Mobile-first" và "API-first". Việc lựa chọn Flutter với kiến trúc BLoC + MVVM cho phép tạo ra một ứng dụng di động có hiệu năng cao, trải nghiệm mượt mà và đồng nhất trên cả iOS và Android. Phía backend, việc sử dụng Celery để xử lý tác vụ nền và Django Channels cho WebSocket đảm bảo hệ thống luôn phản hồi nhanh chóng và sẵn sàng cho các tính năng thời gian thực, đáp ứng kỳ vọng của người dùng hiện đại.

Những phân tích và đúc kết trên đã khẳng định rằng thị trường bất động sản trực tuyến tuy cạnh tranh nhưng vẫn còn rất nhiều dư địa để đổi mới. Chúng đã trở thành nền tảng vững chắc để nhóm đưa ra các quyết định thiết kế chiến lược cho Landify, sẽ được trình bày chi tiết ở mục tiếp theo.

### Chiến lược Xây dựng và Các Quyết định Thiết kế cho Landify

Từ những phân tích sâu sắc về các "khoảng trống" trên thị trường, một chiến lược phát triển tổng thể đã được định hình cho Landify. Mục tiêu không phải là tạo ra một bản sao của các hệ thống hiện có, mà là xây dựng một hệ sinh thái bất động sản minh bạch, tương tác cao và thông minh. Chiến lược này được hiện thực hóa thông qua một kiến trúc đồng bộ và chặt chẽ giữa hai thành phần chính của hệ thống, nơi mỗi "app" trong Django được thiết kế để giải quyết một bài toán nghiệp vụ cụ thể, đảm bảo hệ thống vừa mạnh mẽ, vừa dễ bảo trì và mở rộng.

Nền tảng của hệ thống Backend là sự tách biệt rõ ràng giữa các khái niệm nghiệp vụ, nơi mỗi "app" trong Django được thiết kế để giải quyết một bài toán nghiệp vụ cụ thể, đảm bảo hệ thống vừa mạnh mẽ, vừa dễ bảo trì và mở rộng.

* ***apps.common*** đóng vai trò là hạt nhân, cung cấp các thành phần nền tảng dùng chung. Các file trong app này được thiết kế để tối đa hóa khả năng tái sử dụng và đảm bảo tính nhất quán.
  + models.py định nghĩa BaseModel, giúp chuẩn hóa việc theo dõi thời gian trên toàn bộ CSDL.
  + authentication.py và perms.py tập trung hóa logic xác thực và phân quyền, giúp các app khác có thể dễ dàng áp dụng các quy tắc bảo mật mà không cần lặp lại mã.
  + Thư mục utils/ chứa các hàm tiện ích đã được đóng gói, ví dụ firebase.py để giao tiếp với Firebase Admin SDK, hay sms.py để tương tác với API của Twilio.
* ***apps.users*** được xây dựng để giải quyết trụ cột chiến lược đầu tiên: Xây dựng Niềm tin.
  + models.py định nghĩa User và UserProfile, tách biệt thông tin xác thực cốt lõi khỏi các dữ liệu hồ sơ mở rộng. Quyết định thiết kế này giúp tối ưu hóa các truy vấn liên quan đến đăng nhập. Quan trọng nhất, model User chứa các cờ trạng thái như is\_identity\_verified, là kết quả cuối cùng của quy trình eKYC.
  + services.py chứa các logic nghiệp vụ như change\_user\_avatar, giúp views.py chỉ tập trung vào việc điều phối luồng request-response.
* ***apps.verification*** là nơi hiện thực hóa các giải pháp công nghệ cao để củng cố niềm tin.
  + services.py là trái tim của app này, đóng gói các lời gọi API phức tạp đến các dịch vụ bên ngoài. Thay vì tích hợp trực tiếp logic này vào apps.users, việc tách ra một app riêng giúp dễ dàng quản lý và thay thế các nhà cung cấp dịch vụ trong tương lai. Ví dụ, hàm process\_id\_card\_verification chịu trách nhiệm giao tiếp với microservice OCR, trong khi complete\_ekyc\_liveness\_check làm việc với microservice xác thực khuôn mặt.
  + views.py cung cấp các ViewSet không theo mô hình CRUD (VerificationViewSet, AgoraTokenViewSet), được thiết kế đặc biệt cho các quy trình gồm nhiều bước hoặc các tác vụ tiện ích.
* ***apps.properties và apps.listings*** là cặp đôi trung tâm, hiện thực hóa nghiệp vụ cốt lõi của nền tảng. Quyết định tách biệt "Tài sản" (vật lý, cố định) và "Tin đăng" (thông tin rao bán, có tính thời điểm) là một lựa chọn kiến trúc quan trọng.
  + Trong apps.properties, file models.py định nghĩa Property và các thuộc tính bất biến của nó như Location (sử dụng PostGIS), PropertyType, và PropertyFeature.
  + Trong apps.listings, file models.py chứa Listing liên kết với Property, cho phép một tài sản có thể được đăng bán nhiều lần mà không cần nhập lại thông tin. serializers.py thể hiện rõ sự tối ưu cho từng tác vụ: ListingCreateSerializer xử lý logic tạo lồng phức tạp, trong khi ListingPreviewSerializer làm phẳng dữ liệu để tối ưu hóa cho việc hiển thị danh sách.
* ***apps.interactions*** được thiết kế để giải quyết trụ cột chiến lược thứ hai: Tối ưu hóa Tương tác. App này biến các kênh giao tiếp thành công cụ hỗ trợ giao dịch.
  + models.py: Chat và Message là nền tảng cho hệ thống real-time. Việc sử dụng GenericForeignKey trong model Message là một quyết định thiết kế then chốt, cho phép một tin nhắn có thể trở thành một "Thẻ tương tác" đa năng, liên kết đến Appointment (Lịch hẹn) hoặc Cooperation (Hợp tác).
  + consumers.py và routing.py là nơi hiện thực hóa logic WebSocket với Django Channels, xử lý kết nối, xác thực, và truyền tin nhắn hai chiều theo thời gian thực.
* ***apps.social và apps.moderation*** là hai app bổ trợ, giúp xây dựng một cộng đồng lành mạnh và an toàn.
  + apps.social với các file models.py, views.py, serializers.py cung cấp đầy đủ một mạng xã hội thu nhỏ, cho phép người dùng đăng bài, bình luận, và tương tác.
  + apps.moderation cung cấp một quy trình kiểm duyệt khép kín. models.py định nghĩa luồng Report -> ModerationAction -> Protest, đảm bảo mọi hành vi và quyết định đều được ghi lại một cách minh bạch.
* Cuối cùng, để hiện thực hóa trụ cột chiến lược thứ ba - Cá nhân hóa Trải nghiệm, hệ thống đã được thiết kế để tích hợp dễ dàng với các dịch vụ thông minh.
  + *Microservice Chatbot tư vấn Phong thủy* được xây dựng độc lập, giao tiếp với backend chính thông qua API. apps.verification/services.py chứa hàm get\_feng\_shui\_analysis, đóng vai trò là cầu nối gọi đến microservice này. Kiến trúc này đảm bảo logic AI phức tạp không ảnh hưởng đến hiệu năng của các API cốt lõi.

Để kiến trúc backend mạnh mẽ này có thể được chuyển hóa thành một trải nghiệm người dùng mượt mà, trực quan và dễ dàng bảo trì, phía frontend của Landify (xây dựng bằng Flutter) cũng được thiết kế với một chiến lược kiến trúc rõ ràng và nhất quán. Nền tảng của ứng dụng di động là sự kết hợp giữa hai mẫu kiến trúc hiện đại: BLoC (Business Logic Component) để quản lý trạng thái và luồng dữ liệu, và MVVM (Model-View-ViewModel) để phân tách logic trình bày khỏi giao diện.

Sự tương ứng giữa cấu trúc backend và frontend được thể hiện rõ nét qua cách tổ chức mã nguồn:

* ***Thư mục core/*** đóng vai trò tương tự như apps.common ở backend, là nơi chứa đựng các thành phần nền tảng, có thể tái sử dụng trên toàn bộ ứng dụng.
  + api/: Chứa ApiClient và ApiPaths, định nghĩa cách ứng dụng giao tiếp với các endpoint do Django cung cấp.
  + auth/: Chứa AuthBloc, là BLoC toàn cục quản lý trạng thái đăng nhập của người dùng, tương tác trực tiếp với AuthenticationRepository và UserProfileRepository.
  + models/: Đây là tầng Data Model, chứa các lớp Dart ánh xạ 1-1 với cấu trúc JSON trả về từ API (ví dụ: ListingPreviewModel.dart). Chúng chỉ có nhiệm vụ duy nhất là parse dữ liệu.
  + view\_models/: Đây là tầng ViewModel, nhận Data Model và "chế biến" dữ liệu thành định dạng sẵn sàng cho UI hiển thị (ví dụ: ListingPreviewViewModel.dart định dạng giá tiền, ngày đăng).
  + widgets/: Chứa các widget có thể tái sử dụng ở nhiều màn hình, như PropertyCard hay CustomCard.
* ***Thư mục features/*** được tổ chức tương tự như các "app" của Django, nơi mỗi thư mục con chịu trách nhiệm cho một tính năng hoàn chỉnh, từ giao diện, logic đến việc gọi dữ liệu.
  + ***features/login, signup, otp:*** Là các module giao diện người dùng, tương tác trực tiếp với các BLoC như LoginPhoneBloc, OtpVerificationBloc để thực hiện luồng xác thực do apps.users và Firebase cung cấp.
  + ***features/listing\_create:*** Là một feature phức tạp, phản ánh trực tiếp nghiệp vụ của apps.listings và apps.properties. CreateListingBloc quản lý trạng thái của một form đa bước, thu thập dữ liệu và đóng gói thành ListingCreatePayloadModel, có cấu trúc tương ứng hoàn hảo với ListingCreateSerializer ở backend.
  + ***features/listing\_detail:*** Là nơi tiêu thụ dữ liệu từ nhiều app backend. ListingDetailBloc gọi đến các Repository tương ứng để lấy dữ liệu, sau đó ListingDetailViewModel sẽ tổng hợp và "chế biến" tất cả thông tin thành các ViewModel con (HeaderViewModel, SellerCardViewModel...) để các widget có thể hiển thị một cách độc lập.
  + ***features/chat\_and\_call:*** Hiện thực hóa trụ cột Tối ưu hóa Tương tác. ChatBloc sử dụng đồng thời ChatRepository (để gọi REST API khởi tạo phòng chat) và ChatWebSocketService (để kết nối WebSocket real-time), phản ánh chính xác kiến trúc hybrid của apps.interactions ở backend.
  + ***features/moderation:*** Cung cấp giao diện người dùng cho hệ thống kiểm duyệt. ReportBloc và ProtestBloc giúp người dùng dễ dàng tạo báo cáo hoặc kháng nghị, tương tác trực tiếp với các API do apps.moderation cung cấp.

Vì vậy ta có thể kết luận rằng: Kiến trúc của Landify là một thể thống nhất, không phải là một sự sắp xếp ngẫu nhiên, mà là kết quả của một chiến lược thiết kế có chủ đích, nơi mỗi app ở backend đều có một hoặc nhiều feature tương ứng ở frontend. Sự phân tách rõ ràng các tầng (Data, Logic, Presentation) ở cả hai phía không chỉ là một lựa chọn kỹ thuật, mà là một quyết định chiến lược nhằm tạo ra một hệ thống đồng bộ, dễ bảo trì, và có khả năng mở rộng mạnh mẽ trong tương lai.

## Kiến trúc hệ thống

Hệ thống Landify được xây dựng trên kiến trúc Client-Server phân tán, với sự tách biệt rõ ràng giữa các thành phần frontend, backend và các dịch vụ của bên thứ ba.

Diễn giải các thành phần trong kiến trúc hệ thống & cơ chế hoạt động:

Ảnh có chứa văn bản, ảnh chụp màn hình, biểu đồ, hàng

Nội dung do AI tạo ra có thể không chính xác.

Hình . Sơ đồ kiến trúc tổng quan của hệ thống Landify

Các thành phần chính trong kiến trúc hệ thống bao gồm:

Backend (Server-Side): Là trung tâm xử lý của hệ thống, được xây dựng bằng Python với framework Django và Django Rest Framework (DRF). Chịu trách nhiệm xử lý toàn bộ logic nghiệp vụ, xác thực, và cung cấp RESTful API. Dữ liệu được lưu trữ và quản lý bởi hệ quản trị cơ sở dữ liệu PostgreSQL tích hợp PostGIS để xử lý thông tin địa lý. Các tác vụ nặng và bất đồng bộ được xử lý bởi Celery và Redis.

Frontend (Client-Side): Là ứng dụng di động đa nền tảng, được xây dựng bằng ngôn ngữ Dart và framework Flutter. Đây là giao diện chính mà người dùng tương tác. Ứng dụng áp dụng kiến trúc BLoC (Business Logic Component) để quản lý trạng thái và luồng dữ liệu một cách hiệu quả.

Dịch vụ của bên thứ ba (Third-Party Services): Để tăng tốc độ phát triển và tận dụng các công nghệ chuyên biệt, dự án đã tích hợp nhiều dịch vụ mạnh mẽ:

* Firebase: Được sử dụng cho việc xác thực người dùng (Authentication), các tính năng thời gian thực như Chat (Cloud Firestore), và cấu hình động cho ứng dụng (Remote Config).
* Cloudinary: Chịu trách nhiệm lưu trữ, tối ưu và phân phối toàn bộ media (hình ảnh, video) của dự án.
* Các API chuyên dụng: Tích hợp FPT.AI cho tính năng eKYC, Twilio cho việc gửi SMS OTP, và Agora cho tính năng gọi video.

Sự giao tiếp giữa ứng dụng Flutter (Client) và hệ thống Django (Server) được thực hiện hoàn toàn thông qua RESTful API theo chu trình Yêu cầu – Phản hồi.

Khi người dùng thực hiện một hành động trên ứng dụng Flutter, một yêu cầu HTTP (ví dụ GET, POST) sẽ được tạo ra và gửi đến endpoint API tương ứng trên server Django. Server nhận yêu cầu, tiến hành xác thực (nếu cần), sau đó xử lý logic nghiệp vụ và tương tác với cơ sở dữ liệu PostgreSQL. Dữ liệu sau khi được xử lý sẽ được định dạng thành JSON thông qua các lớp Serializer và gửi ngược lại cho ứng dụng Flutter trong một phản hồi HTTP. Ứng dụng Flutter nhận dữ liệu JSON này, phân tích thành các đối tượng dữ liệu và cập nhật lại giao diện người dùng để hiển thị thông tin mới.

Đối với các tính năng thời gian thực như Chat, luồng hoạt động có sự tham gia của Firestore. Client vẫn gọi API Django để khởi tạo cuộc trò chuyện, nhưng sau đó sẽ lắng nghe trực tiếp các thay đổi trên một document của Firestore để nhận và gửi tin nhắn mới, tạo ra trải nghiệm tức thời cho người dùng.

## Phân tích hệ thống

Ảnh có chứa văn bản, ảnh chụp màn hình, biểu đồ, vòng tròn

Nội dung do AI tạo ra có thể không chính xác.

Trong hệ thống Landify, "Người dùng" là tác nhân cơ bản nhất, đại diện cho bất kỳ cá nhân nào tương tác với ứng dụng, bao gồm cả khách truy cập chưa có tài khoản và những người dùng đang trong quá trình đăng ký. Các chức năng được thiết kế cho tác nhân này tập trung vào việc cung cấp giá trị ngay từ lần truy cập đầu tiên và xây dựng một luồng gia nhập hệ thống liền mạch.

* ***Khám phá và Tìm kiếm thông tin:*** Ngay cả khi chưa đăng nhập, Người dùng vẫn có thể truy cập vào các tính năng khám phá cốt lõi của nền tảng.
  + Tìm kiếm & Lọc BĐS: Đây là chức năng mở, cho phép bất kỳ ai cũng có thể tìm kiếm và lọc danh sách bất động sản theo các tiêu chí cơ bản như vị trí, loại hình, khoảng giá.
  + Xem chi tiết BĐS: Từ danh sách tìm kiếm, người dùng có thể nhấn vào để xem thông tin chi tiết của một bất động sản. Chức năng này bao hàm (<<include>>) khả năng Xem phân tích dữ liệu, ví dụ như biểu đồ biến động giá hoặc các thông tin phân tích thị trường liên quan đến bất động sản đó.
  + Lưu tin đăng: Người dùng có thể lưu lại các tin đăng mà họ quan tâm để xem lại sau. Hệ thống sẽ yêu cầu đăng nhập/đăng ký nếu hành động này được thực hiện bởi một khách truy cập.
* ***Quản lý Tài khoản và Định danh:*** Đây là luồng chức năng nền tảng để một khách truy cập trở thành một thành viên của cộng đồng Landify.
  + Đăng ký: Cho phép người dùng tạo một tài khoản mới. Quy trình này bắt buộc (<<include>>) người dùng phải thực hiện Xác thực mã OTP qua số điện thoại để đảm bảo tính chính chủ. Sau khi xác thực thành công, hệ thống sẽ yêu cầu người dùng Cập nhật thông tin cá nhân cơ bản (<<include>>), chẳng hạn như họ và tên, để hoàn tất hồ sơ ban đầu.
  + Đăng nhập: Dành cho người dùng đã có tài khoản. Tương tự như đăng ký, luồng đăng nhập bằng số điện thoại cũng bao hàm (<<include>>) bước Xác thực mã OTP để tăng cường bảo mật.
  + Xem trang cá nhân: Sau khi đăng nhập, người dùng có thể truy cập trang cá nhân của mình. Từ đây, họ có thể thực hiện hành động Thay đổi ảnhđại diện, một chức năng mở rộng (<<extend>>) giúp cá nhân hóa hồ sơ.
* ***Xác thực và Nâng cao Độ tin cậy:*** Để có thể sử dụng các tính năng giao dịch quan trọng (như đăng tin), người dùng cần phải nâng cao độ tin cậy của tài khoản.
  + Xác thực danh tính: Đây là một quy trình eKYC toàn diện, được tổng hợp từ hai bước nhỏ hơn là Xác thực CCCD (chụp và trích xuất thông tin từ Căn cước công dân) và Xác thực khuôn mặt (so khớp khuôn mặt với ảnh trên CCCD). Việc hoàn thành cả hai bước này là điều kiện tiên quyết để một "Người dùng" được cấp đầy đủ quyền của một "Thành viên" trên hệ thống.

Thông qua các chức năng này, tác nhân "Người dùng" được dẫn dắt qua một hành trình rõ ràng: từ việc khám phá thông tin, tạo và bảo mật tài khoản, cho đến việc hoàn tất xác thực danh tính để trở thành một thành viên đáng tin cậy của cộng đồng Landify.

Ảnh có chứa văn bản, ảnh chụp màn hình, vòng tròn, biểu đồ

Nội dung do AI tạo ra có thể không chính xác.

Hình . Sơ đồ Use Case – Actor: Thành viên

Sau khi hoàn tất quá trình đăng ký và đăng nhập vào hệ thống, người dùng sẽ đảm nhận vai trò "Thành viên". Đây là tác nhân trung tâm, là hạt nhân của mọi hoạt động tương tác và giao dịch trên nền tảng Landify. Với vai trò này, người dùng được cấp quyền truy cập vào một hệ sinh thái tính năng toàn diện, được phân chia thành các nhóm nghiệp vụ chính như sau:

* ***Quản lý Bất động sản và Tin đăng:*** Đây là nhóm chức năng cốt lõi, cho phép Thành viên tham gia trực tiếp vào thị trường.
  + Đăng tin nhà đất: Là chức năng trung tâm, cho phép người dùng đăng tải thông tin bất động sản của mình. Quy trình này được làm giàu bởi các tính năng tích hợp (<<include>>) như Mua gói VIP để tăng hiệu quả hiển thị và Áp dụng khuyến mãi để tối ưu chi phí.
  + Quản lý tin đăng: Sau khi đăng, Thành viên có toàn quyền truy cập để chỉnh sửa, cập nhật trạng thái (ví dụ: "đã bán"), hoặc xóa các tin đăng của chính mình.
* ***Tương tác và Giao tiếp Trực tiếp:*** Nhóm chức năng này tập trung vào việc phá bỏ rào cản giao tiếp, tạo ra một kênh kết nối liền mạch và hiệu quả.
  + Chat trực tiếp: Là công cụ giao tiếp chính, cho phép người mua và người bán trao đổi thông tin, thương lượng một cách nhanh chóng. Chức năng này được mở rộng (<<include>>) với khả năng thực hiện Gọi Video Call trực tiếp trong giao diện chat, hỗ trợ việc xem nhà từ xa.
  + Đặt lịch hẹn xem nhà: Một công cụ nghiệp vụ cho phép người mua gửi yêu cầu xem nhà chính thức đến người bán với thời gian và địa điểm cụ thể.
  + Hợp tác môi giới: Tính năng chuyên biệt cho phép các nhà môi giới gửi lời đề nghị hợp tác chính thức đến chủ tin đăng để cùng phân phối bất động sản.
* ***Xây dựng Cộng đồng và Kiểm duyệt:*** Các chức năng này trao quyền cho Thành viên, giúp họ cùng chung tay xây dựng một môi trường giao dịch an toàn và minh bạch.
  + Báo cáo tin đăng vi phạm: Cho phép người dùng báo cáo các tin đăng có dấu hiệu lừa đảo, thông tin sai sự thật hoặc vi phạm quy định của nền tảng.
  + Gửi kháng nghị tin đăng: Cung cấp một kênh chính thức để Thành viên có thể khiếu nại khi cho rằng tin đăng của mình bị xử lý không thỏa đáng.
  + Gửi đánh giá nhà đất: Sau một giao dịch hoặc tương tác, Thành viên có thể để lại đánh giá về bất động sản hoặc người bán, giúp những người dùng sau có thêm thông tin tham khảo.
  + Mạng xã hội: Tham gia vào một không gian cộng đồng, nơi các thành viên có thể chia sẻ kinh nghiệm, kiến thức và thảo luận về các chủ đề liên quan đến bất động sản.
* ***Cá nhân hóa và Theo dõi:*** Nhóm chức năng giúp cá nhân hóa trải nghiệm và giữ cho người dùng luôn được cập nhật thông tin.
  + Theo dõi người bán: Cho phép Thành viên theo dõi hoạt động của những nhà môi giới hoặc chủ nhà mà họ tin tưởng.
  + Nhận thông báo: Chức năng này có thể được mở rộng (<<extend>>) từ việc theo dõi. Khi một người bán được theo dõi có hoạt động mới (ví dụ: đăng tin mới), hệ thống sẽ tự động gửi thông báo đến cho Thành viên.

Thông qua các nhóm chức năng này, tác nhân "Thành viên" không chỉ là người tiêu thụ thông tin mà còn là người tạo ra giá trị, thúc đẩy các tương tác và góp phần quan trọng vào việc xây dựng một hệ sinh thái bất động sản năng động và đáng tin cậy.

Ảnh có chứa văn bản, ảnh chụp màn hình, biểu đồ, hàng

Nội dung do AI tạo ra có thể không chính xác.

Bên cạnh các tác nhân là người dùng cuối, hệ thống Landify còn được vận hành và kiểm soát bởi hai tác nhân quan trọng khác: "Admin" (Quản trị viên) và "Hệ thống" (Các quy trình tự động). Hai tác nhân này đảm bảo cho nền tảng hoạt động một cách trơn tru, an toàn và thông minh.

Tác nhân "Admin" (Quản trị viên)

Admin là người có quyền hạn cao nhất, chịu trách nhiệm giám sát và quản lý toàn bộ hoạt động trên nền tảng. Các chức năng của Admin tập trung vào việc duy trì chất lượng nội dung và đảm bảo an toàn cho cộng đồng.

* ***Quản lý Nội dung và Tin đăng:***
  + *Kiểm duyệt tin đăng:* Admin có quyền xem xét các tin đăng mới, phê duyệt các tin hợp lệ, hoặc từ chối và gỡ bỏ các tin đăng vi phạm chính sách.
  + *Đẩy tin tiềm năng:* Dựa trên các phân tích hoặc chiến dịch marketing, Admin có thể chủ động đẩy các tin đăng nổi bật, tiềm năng lên các vị trí ưu tiên để tăng khả năng tiếp cận.
* ***Quản lý Người dùng:***
  + Quản lý người dùng: Đây là một chức năng tổng quát, cho phép Admin xem danh sách, tìm kiếm và truy cập thông tin chi tiết của tất cả người dùng trên hệ thống. Chức năng này được mở rộng (<<extend>>) với các hành động cụ thể hơn như:
    - Thêm/sửa/xóa người dùng: Cho phép Admin tạo tài khoản mới, chỉnh sửa thông tin hoặc xóa vĩnh viễn một tài khoản khỏi hệ thống trong những trường hợp cần thiết.
    - Vô hiệu hóa tài khoản vi phạm: Thay vì xóa vĩnh viễn, đây là một hành động mềm hơn (<<extend>>), cho phép Admin tạm thời khóa hoặc mở lại tài khoản của một thành viên dựa trên mức độ vi phạm.
* ***Xử lý Vi phạm và Khiếu nại:***
  + Xử lý báo cáo kháng nghị: Khi nhận được các báo cáo vi phạm từ Thành viên hoặc các đơn kháng nghị về quyết định kiểm duyệt, Admin là người có trách nhiệm xem xét và đưa ra phán quyết cuối cùng, đảm bảo tính công bằng và minh bạch cho quy trình.

Tác nhân "Hệ thống" (System)

"Hệ thống" là một tác nhân phi con người, đại diện cho các quy trình và thuật toán tự động chạy ở phía backend. Các chức năng này được thiết kế để tự động hóa các tác vụ, phân tích dữ liệu và cung cấp các tính năng thông minh mà không cần sự can thiệp trực tiếp của con người.

* ***Tự động hóa Kiểm duyệt và Bảo mật:***
  + *Phát hiện Spam/Scam:* Ngay sau khi một tin đăng được tạo, Hệ thống sẽ tự động kích hoạt một tác vụ nền (Celery task) để quét nội dung, tiêu đề của tin đăng. Nếu phát hiện các từ khóa bị cấm hoặc các dấu hiệu lừa đảo, tin đăng sẽ tự động bị gỡ bỏ hoặc gắn cờ để Admin xem xét.
  + *Xác minh vị trí người đánh giá:* Để tăng tính xác thực cho các bài đánh giá, Hệ thống có thể tự động kiểm tra và so sánh vị trí địa lý của người dùng tại thời điểm viết đánh giá với vị trí của bất động sản được đánh giá.
* *Phân tích và Hỗ trợ Ra quyết định:*
  + *Tính toán chỉ số phong thủy:* Khi người dùng yêu cầu, Hệ thống sẽ tự động gọi đến microservice chuyên biệt để tính toán bản mệnh dựa trên năm sinh và phân tích độ tương hợp với hướng của bất động sản, sau đó trả về kết quả phân tích chi tiết.
  + *Phân tích & Dự đoán giá:* Dựa trên dữ liệu lớn (Big Data) về các giao dịch trong quá khứ, thông tin vị trí từ PostGIS và các đặc điểm của bất động sản, Hệ thống có khả năng chạy các mô hình học máy (Machine Learning) để phân tích và đưa ra một mức giá tham khảo, giúp người dùng có thêm cơ sở để ra quyết định.

### Đặc tả Use Case

#### Xác thực danh tính (eKYC)

Bảng . Đặc tả UC: Xác thực danh tính (eKYC)

|  |  |
| --- | --- |
| Use case Id | UC001 |
| Name  (Tên use case) | Xác thực danh tính (eKYC) |
| Description  (Mô tả) | Use case này cho phép Người dùng đã đăng nhập thực hiện quy trình xác minh danh tính điện tử (eKYC) hai bước. Bước một bao gồm việc chụp và trích xuất thông tin từ Căn cước công dân (CCCD). Bước hai là quay video khuôn mặt để xác thực người thật và so sánh với ảnh trên CCCD. |
| Primary Actor  (Actor chính) | Người dùng (User) |
| Secondary Actor (Actor phụ) | FPT.AI (Dịch vụ nhận dạng ký tự quang học và xác thực khuôn mặt) |
| Pre-conditions  (Tiền điều kiện) | 1. Người dùng đã đăng nhập thành công vào hệ thống Landify.  2. Người dùng đang ở màn hình lựa chọn phương thức xác thực và đã chọn bắt đầu quy trình. |
| Post-conditions  (Hậu điều kiện) | ***Thành công:***  Trạng thái tài khoản người dùng trong CSDL được cập nhật thành is\_id\_card\_verified = True và is\_identity\_verified = True.  2. Người dùng được cấp quyền truy cập vào các tính năng yêu cầu xác thực (ví dụ: đăng tin, đặt lịch hẹn, đánh giá,…).  3. Hệ thống hiển thị thông báo xác thực thành công.  ***Thất bại:***  1. Trạng thái tài khoản người dùng không thay đổi.  2. Hệ thống hiển thị thông báo lỗi cụ thể. |
| Main flows  (Luồng hoạt động) | ***Bước 1: Quét Căn cước công dân (CCCD)***  1. Use case bắt đầu khi Người dùng chọn "Bắt đầu quét CCCD" trên màn hình xác thực.  2. Hệ thống yêu cầu và kiểm tra quyền truy cập camera.  3. Hệ thống khởi tạo và hiển thị giao diện camera với một khung (overlay) để căn chỉnh CCCD.  4. Người dùng đặt mặt trước của CCCD vào trong khung và nhấn nút chụp ảnh.  5. Ứng dụng gửi tệp ảnh đã chụp đến backend  6. Backend nhận tệp ảnh và gọi đến API FPT.AI ID Recognition để trích xuất thông tin.  7. FPT.AI xử lý và trả về dữ liệu đã trích xuất (số CCCD, họ tên, ngày sinh...).  8. Backend cập nhật trạng thái is\_id\_card\_verified = True cho người dùng và trả về thông báo thành công cho ứng dụng.  9. Ứng dụng hiển thị thông báo thành công cho Bước 1 và tự động chuyển người dùng sang Bước 2.  ***Bước 2: Xác thực khuôn mặt***  10. Hệ thống khởi tạo lại giao diện camera trước với một khung (overlay) hình oval để căn chỉnh khuôn mặt.  11. Hệ thống sử dụng công nghệ Face Detection trên thiết bị để liên tục kiểm tra xem có khuôn mặt trong khung hình hay không.  12. Khi hệ thống phát hiện khuôn mặt nằm ổn định trong khung, một bộ đếm ngược 5 giây sẽ tự động bắt đầu và hệ thống bắt đầu quay video.  13. Sau khi đếm ngược kết thúc, hệ thống tự động dừng quay và có được một tệp video ngắn.  14. Ứng dụng gửi tệp video này đến backend Django.  15. Backend gọi đến API FPT.AI Liveness Detection, gửi kèm video và ảnh CCCD đã được lưu cache từ Bước 1.  16. FPT.AI phân tích video để xác thực người thật (liveness) và so sánh khuôn mặt trong video với ảnh trên CCCD.  17. Nếu cả hai điều kiện đều hợp lệ, FPT.AI trả về kết quả thành công.  18. Backend cập nhật trạng thái is\_identity\_verified = True cho người dùng và trả về phản hồi thành công.  19. Ứng dụng hiển thị thông báo "Xác minh danh tính hoàn tất!" cho người dùng.  20. Use case kết thúc. |
| Alternative flows (Luồng thay thế) | 3a. Người dùng không cấp quyền camera:  1. Tại bước 2, nếu người dùng từ chối cấp quyền camera.  2. Hệ thống hiển thị thông báo yêu cầu cấp quyền để tiếp tục.  3. Use case kết thúc.  12a. Khuôn mặt di chuyển ra khỏi khung:  1. Tại bước 12, nếu khuôn mặt người dùng di chuyển ra khỏi khung oval trong quá trình đếm ngược.  2. Bộ đếm ngược sẽ dừng lại và reset.  3. Hệ thống hiển thị lại hướng dẫn "Vui lòng giữ yên khuôn mặt trong khung".  4. Luồng hoạt động quay lại bước 11.  17a. Xác thực liveness hoặc so khớp khuôn mặt thất bại:  1. Tại bước 17, nếu FPT.AI trả về kết quả không phải người thật hoặc khuôn mặt không khớp.  2. Backend trả về thông báo lỗi cụ thể cho ứng dụng (ví dụ: "Khuôn mặt không khớp với CCCD").  3. Ứng dụng hiển thị lỗi và cho phép người dùng thực hiện lại từ bước 10. |
| Exception flows (Luồng ngoại lệ) | 6a. Không thể đọc thông tin từ CCCD:  1. Tại bước 6, nếu API của FPT.AI không thể trích xuất được thông tin từ ảnh (do ảnh quá mờ, lóa sáng).  2. Hệ thống trả về lỗi "Không thể đọc thông tin từ ảnh CCCD. Vui lòng thử lại với ảnh rõ nét hơn."  3. Luồng hoạt động quay lại bước 3.  7a. API FPT.AI gặp sự cố:  1. Tại bước 7 hoặc 17, nếu dịch vụ của FPT.AI không phản hồi hoặc báo lỗi hệ thống.  2. Backend trả về một thông báo lỗi chung (ví dụ: "Dịch vụ xác thực đang tạm thời gián đoạn. Vui lòng thử lại sau.").  3. Use case kết thúc. |

#### Tạo tin đăng

Bảng . Đặc tả UC: Tạo tin đăng

|  |  |
| --- | --- |
| Use case Id | UC002 |
| Name  (Tên use case) | Tạo tin đăng |
| Description  (Mô tả) | Use case này cho phép Người dùng đã đăng nhập thực hiện quy trình xác minh danh tính điện tử (eKYC) hai bước. Bước một bao gồm việc chụp và trích xuất thông tin từ Căn cước công dân (CCCD). Bước hai là quay video khuôn mặt để xác thực người thật và so sánh với ảnh trên CCCD. |
| Primary Actor  (Actor chính) | Người dùng (User) |
| Secondary Actor (Actor phụ) | Cloudinary |
| Pre-conditions  (Tiền điều kiện) | 1. Người dùng đã đăng nhập vào hệ thống Landify.  2. Tài khoản của người dùng đã hoàn thành xác thực danh tính (eKYC). |
| Post-conditions  (Hậu điều kiện) | **Thành công:**  1. Một bản ghi Property và Listing mới, cùng các bản ghi phụ thuộc (Location, ListingPropertyFeatureValue...) được tạo thành công trong cơ sở dữ liệu.  2. Các tệp media (hình ảnh, video) được tải lên và lưu trữ trên Cloudinary, URL được liên kết với bản ghi Property.  3. Một tác vụ nền được kích hoạt để kiểm tra nội dung tin đăng có chứa spam hay không.  4. Người dùng được điều hướng đến màn hình thông báo đăng tin thành công.  **Thất bại:**  1. Không có bản ghi nào được tạo trong cơ sở dữ liệu.  2. Hệ thống hiển thị thông báo lỗi cụ thể cho người dùng. |
| Main flows  (Luồng hoạt động) | 1. Use case bắt đầu khi Người dùng nhấn vào nút "Đăng tin" trên giao diện chính của ứng dụng.  2. Hệ thống hiển thị giao diện tạo tin đăng, bắt đầu từ ***Bước 1: Thông tin BĐS.***  3. Người dùng chọn loại nhu cầu (Bán/Cho thuê), sau đó nhập các thông tin chi tiết về bất động sản, bao gồm:  - Địa chỉ (thông qua màn hình tìm kiếm hoặc chọn trên bản đồ).  - Loại hình BĐS, diện tích, giá, đơn vị giá.  - Thông tin khác như giấy tờ pháp lý, số phòng ngủ, hướng nhà...  - Tiêu đề và mô tả chi tiết cho tin đăng.  - Thông tin liên hệ.  4. Ứng dụng thực hiện kiểm tra tính đầy đủ của dữ liệu ngay trên thiết bị khi người dùng nhập.  5. Người dùng nhấn nút "Tiếp tục". Hệ thống chuyển sang ***Bước 2: Hình ảnh & Video.***  6. Người dùng chọn và tải lên các hình ảnh từ thư viện hoặc chụp ảnh mới. Ứng dụng cho phép tải lên tối thiểu 3 ảnh.  7. Người dùng có thể sắp xếp lại thứ tự ảnh, đặt một ảnh làm ảnh đại diện, và thêm mô tả cho từng ảnh.  8. Người dùng có thể tùy chọn dán đường dẫn video từ YouTube hoặc TikTok.  9. Người dùng nhấn nút "Tiếp tục". Hệ thống chuyển sang ***Bước 3: Hoàn tất.***  10. Hệ thống hiển thị danh sách các loại tin đăng (ví dụ: Tin thường, VIP Bạc, VIP Vàng...).  11. Người dùng chọn một loại tin đăng. Ứng dụng hiển thị màn hình cấu hình chi tiết cho phép chọn số ngày đăng và ngày bắt đầu.  12. Người dùng có thể tùy chọn áp dụng mã khuyến mãi.  13. Hệ thống hiển thị tóm tắt chi phí cuối cùng.  14. Người dùng nhấn nút "Đăng tin".  15. Ứng dụng tổng hợp toàn bộ dữ liệu từ 3 bước.  16. Ứng dụng gửi một yêu cầu đến server, đính kèm payload và các tệp hình ảnh  17. Backend Django nhận yêu cầu và xử lý nghiệp.  18. Trong một yêu cầu tạo tin đăng, backend sẽ:  a. Tải các tệp media lên dịch vụ Cloudinary.  b. Tạo các bản ghi Location, Property mới.  c. Tạo bản ghi Listing chính và các bản ghi chi tiết liên quan (BuySellDetail, RentalDetail...).  19. Nếu tất cả các bước trên thành công. Backend kích hoạt một tác vụ nền Celery và trả về phản hồi thành công kèm theo dữ liệu của tin đăng vừa tạo.  20. Ứng dụng nhận được phản hồi thành công và điều hướng Người dùng đến màn hình "Đăng tin thành công".  21. Use case kết thúc. |
| Alternative flows (Luồng thay thế) | 5a. Người dùng muốn quay lại bước trước:  1. Tại Bước 2 hoặc Bước 3, người dùng nhấn nút "Quay lại".  2. Hệ thống chuyển về giao diện của bước trước đó, giữ nguyên các dữ liệu người dùng đã nhập.  15a. Dữ liệu không hợp lệ ở phía Client:  1. Tại bất kỳ bước nào, nếu người dùng nhấn "Tiếp tục" nhưng chưa điền đủ các trường bắt buộc.  2. Ứng dụng sẽ hiển thị thông báo lỗi ngay trên các trường bị thiếu và không cho phép chuyển sang bước tiếp theo cho đến khi dữ liệu hợp lệ. |
| Exception flows (Luồng ngoại lệ) | 18a. Lỗi trong quá trình xử lý ở Backend:  1. Tại bước 18, nếu có bất kỳ lỗi nào xảy ra (ví dụ: kiểm tra dữ liệu đầu vào thất bại, lỗi kết nối đến Cloudinary, lỗi ghi CSDL).  2. Phía Backend đảm bảo cơ sở dữ liệu đảm bảo không có dữ liệu nào được lưu lại một phần.  3. Backend trả về một phản hồi lỗi HTTP (ví dụ: 400 Bad Request) kèm theo thông báo lỗi chi tiết.  4. Ứng dụng hiển thị thông báo lỗi cho người dùng.  5. Use case kết thúc trong trạng thái thất bại. |

#### Hợp tác môi giới

Bảng . Đặc tả UC: Hợp tác môi giới

|  |  |
| --- | --- |
| Use case Id | UC003 |
| Name  (Tên use case) | Hợp tác môi giới |
| Description  (Mô tả) | Use case này mô tả quy trình một Người dùng (Nhà môi giới) gửi một lời đề nghị hợp tác chính thức đến Chủ tin đăng ngay bên trong giao diện trò chuyện (chat). Hệ thống sẽ tạo một bản ghi yêu cầu ở backend và đồng thời gửi một tin nhắn đặc biệt vào cuộc trò chuyện trên Firestore để Chủ tin đăng có thể phản hồi |
| Primary Actor  (Actor chính) | Người dùng (User) - trong vai trò Nhà môi giới và Chủ tin đăng |
| Secondary Actor (Actor phụ) | Cloud Firestore |
| Pre-conditions  (Tiền điều kiện) | 1. Cả Nhà môi giới và Chủ tin đăng đều đã đăng nhập vào hệ thống.  2. Nhà môi giới đã bắt đầu một cuộc trò chuyện với Chủ tin đăng về một tin đăng cụ thể. |
| Post-conditions  (Hậu điều kiện) | **Thành công:**  1. Một bản ghi Cooperation được tạo trong CSDL PostgreSQL với trạng thái PENDING.  2. Một tin nhắn đặc biệt (loại cooperation\_request) được tạo trong cuộc trò chuyện tương ứng.  3. Trạng thái của bản ghi Cooperation được cập nhật thành ACCEPTED hoặc REJECTED sau khi Chủ tin đăng phản hồi.  4. Giao diện tin nhắn đặc biệt trên cả hai thiết bị được cập nhật để phản ánh trạng thái mới  **Thất bại:**  1. Yêu cầu hợp tác không được tạo.  2. Hệ thống hiển thị thông báo lỗi. |
| Main flows  (Luồng hoạt động) | ***Phần 1: Nhà môi giới gửi yêu cầu từ trong Chat***  1. Use case bắt đầu khi Nhà môi giới đang trong màn hình chat với Chủ tin đăng, nhấn vào nút menu mở rộng (dấu cộng).  2. Hệ thống hiển thị danh sách các hành động, bao gồm "Ngỏ lời hợp tác môi giới".  3. Nhà môi giới chọn "Ngỏ lời hợp tác môi giới".  4. Ứng dụng Flutter gửi một yêu cầu đến server kèm theo thông tin đăng liên quan đến cuộc trò chuyện.  5. Backend Django nhận yêu cầu và xử lý logic:  a. Xác định agent (người yêu cầu) và owner (chủ tin đăng).  b. Kiểm tra các điều kiện nghiệp vụ (không gửi yêu cầu trùng lặp...).  6. Nếu hợp lệ, backend tạo một bản ghi Cooperation mới trong CSDL với trạng thái chờ xác nhận.  7. Backend trả về phản hồi thành công chứa dữ liệu của yêu cầu vừa tạo.  8. Đồng thời, sau khi nhận được phản hồi thành công từ backend, ứng dụng của Nhà môi giới sẽ gọi đến Cloud Firestore để thêm một tin nhắn mới vào cuộc trò chuyện. Tin nhắn này có dạng đặc biệt, tham chiếu đến bản ghi Cooperation vừa tạo ở bước 6.  9. Giao diện chat của cả hai người dùng ngay lập tức hiển thị một thẻ "Đề nghị hợp tác" đang ở trạng thái "Chờ phản hồi".  ***Phần 2: Chủ tin đăng phản hồi ngay trên giao diện*** ***Chat***  10. Chủ tin đăng nhìn thấy thẻ "Đề nghị hợp tác" trong cuộc trò chuyện và nhấn vào nút "Chấp thuận" hoặc "Từ chối" ngay trên thẻ đó.  11. Ứng dụng của Chủ tin đăng gửi một yêu cầu đến server, kèm theo thông tin chấp nhận hay từ chối yêu cầu hợp tác này.  12. Backend Django nhận yêu cầu, xác thực quyền và cập nhật trạng thái của bản ghi lời mời hợp tác tương ứng  13. Backend kích hoạt tác vụ nền để gửi thông báo đến Nhà môi giới.  14. Backend trả về phản hồi thành công.  15. Nhờ cơ chế real-time của Firestore, giao diện thẻ "Đề nghị hợp tác" trên cả hai máy của người dùng sẽ tự động cập nhật sang trạng thái "Đã chấp nhận" hoặc "Đã từ chối" mà không cần tải lại.  16. Use case kết thúc. |
| Alternative flows (Luồng thay thế) | 4a. Gửi yêu cầu không hợp lệ:  1. Tại bước 4, nếu backend xác định yêu cầu không hợp lệ (ví dụ: đã có yêu cầu đang chờ).  2. Backend trả về lỗi 400 Bad Request.  3. Ứng dụng của Nhà môi giới sẽ không gửi tin nhắn lên Firestore và thay vào đó hiển thị một thông báo lỗi (ví dụ: "Bạn đã gửi yêu cầu này trước đó").  10a. Chủ tin đăng không phản hồi qua Chat:  1. Tại bước 10, nếu Chủ tin đăng không tương tác với thẻ trong chat mà vào khu vực quản lý riêng.  2. Chủ tin đăng vẫn có thể phản hồi yêu cầu thông qua danh sách quản lý chung (như mô tả trong luồng UC-05 trước đó). Hệ thống vẫn hoạt động bình thường, tuy nhiên tin nhắn trong chat sẽ không được cập nhật trạng thái. |
| Exception flows (Luồng ngoại lệ) | 8a. Lỗi kết nối đến Firestore:  1. Tại bước 8, sau khi đã tạo yêu cầu thành công ở backend nhưng ứng dụng gặp lỗi kết nối và không thể gửi tin nhắn lên Firestore.  2. Yêu cầu hợp tác vẫn tồn tại ở backend và có thể được xử lý ở khu vực quản lý riêng, nhưng sẽ không hiển thị trong cuộc trò chuyện.  12a. Người không có thẩm quyền cố gắng phản hồi:  1. Tại bước 12, nếu một người dùng không phải là Chủ tin đăng (ví dụ: Nhà môi giới) cố gắng giả mạo yêu cầu phản hồi.  2. Backend sẽ kiểm tra quyền và trả về lỗi ngăn cấm.  3. Use case kết thúc. |

## Thiết kế hệ thống

### Sơ đồ hoạt động

#### Xác thực danh tính

Ảnh có chứa văn bản, ảnh chụp màn hình, Song song, biểu đồ

Nội dung do AI tạo ra có thể không chính xác.

Hình . Sơ đồ hoạt động: Xác thực danh tính (eKYC)

#### Tạo tin đăng

Ảnh có chứa văn bản, biểu đồ, ảnh chụp màn hình, Song song

Nội dung do AI tạo ra có thể không chính xác.

Hình . Sơ đồ hoạt động: Tạo tin đăng

#### Hợp tác môi giới

Ảnh có chứa văn bản, ảnh chụp màn hình, biểu đồ, Song song

Nội dung do AI tạo ra có thể không chính xác.

Hình . Sơ đồ hoạt động: Hợp tác môi giới

### Thiết kế cơ sở dữ liệu

Để đảm bảo tính rõ ràng và dễ bảo trì, lược đồ cơ sở dữ liệu của hệ thống Landify được thiết kế theo từng cụm chức năng (domain) riêng biệt. Cách tiếp cận này giúp làm nổi bật các mối quan hệ nghiệp vụ quan trọng trong từng phân hệ, thay vì trình bày một sơ đồ tổng thể phức tạp.

#### Cụm Người dùng và Tương tác Xã hội

Cụm này là trung tâm của hệ thống, định nghĩa thực thể người dùng và cách họ tương tác với nhau cũng như với các tính năng xã hội và quảng cáo.

Ảnh có chứa văn bản, ảnh chụp màn hình, biểu đồ, Hình chữ nhật

Nội dung do AI tạo ra có thể không chính xác.

Giải thích các thực thể và mối quan hệ chính:

* ***User (Người dùng):***
  + Vai trò: Đây là thực thể trung tâm, kế thừa từ AbstractUser của Django để lưu trữ thông tin xác thực cốt lõi.
  + Các thuộc tính quan trọng: Ngoài các trường cơ bản, User chứa các cờ boolean quan trọng như is\_phone\_verified và is\_identity\_verified để quản lý trạng thái xác thực, và role để phân quyền giữa người dùng thường và quản trị viên.
* ***UserProfile (Hồ sơ Người dùng):***
  + Mối quan hệ: Một-Một với User. Mỗi người dùng chỉ có một hồ sơ duy nhất.
  + Vai trò: Quyết định thiết kế tách UserProfile ra khỏi User nhằm mục đích tối ưu hóa. Bảng User chính được giữ gọn nhẹ, chỉ chứa các thông tin thường xuyên được truy vấn khi xác thực. Trong khi đó, UserProfile lưu trữ các thông tin mở rộng ít thay đổi hơn như ảnh đại diện (avatar), ngày sinh, giới thiệu, và các chỉ số thống kê như rating\_score (điểm đánh giá) và listing\_count (số tin đăng).
* ***Subscription (Theo dõi):***
  + Mối quan hệ: Bảng trung gian này hiện thực hóa mối quan hệ Nhiều-Nhiều giữa các User với chính nó.
  + Vai trò: Mỗi bản ghi trong Subscription đại diện cho một hành động "theo dõi", với follower là người thực hiện và following là người được theo dõi. Cấu trúc này cho phép xây dựng các tính năng mạng xã hội như "danh sách người theo dõi" và "danh sách đang theo dõi".
* ***UserPromotion (Khuyến mãi của Người dùng):***
  + Mối quan hệ: Nhiều-Một với User. Một người dùng có thể nhận được nhiều mã khuyến mãi khác nhau.
  + Vai trò: Lưu trữ một phiên bản cụ thể của một chương trình khuyến mãi đã được cấp cho người dùng, bao gồm mã code duy nhất, ngày hết hạn (expiry\_date), và trạng thái sử dụng. Bảng này cũng có một liên kết tùy chọn (used\_on\_listing) đến tin đăng mà mã đã được áp dụng.
* ***Chat (Cuộc trò chuyện):***
  + Mối quan hệ: Nhiều-Nhiều với User thông qua trường participants.
  + Vai trò: Đại diện cho một phòng chat, có thể là PRIVATE (giữa hai người) hoặc GROUP. Bảng này lưu trữ những người tham gia và thời gian của tin nhắn cuối cùng (last\_message\_timestamp) để dễ dàng sắp xếp danh sách các cuộc trò chuyện.
* ***Message (Tin nhắn):***
  + Mối quan hệ: Nhiều-Một với Chat. Mỗi tin nhắn chỉ thuộc về một cuộc trò chuyện.
  + Vai trò: Đây là thực thể cốt lõi của tính năng chat.
  + Thiết kế nổi bật (GenericForeignKey): Thay vì chỉ lưu trữ văn bản, Message được thiết kế để trở thành một phương tiện truyền tải các đối tượng nghiệp vụ. Bằng cách sử dụng cặp trường content\_type (ID của loại model) và object\_id (ID của bản ghi), một tin nhắn có thể liên kết đến bất kỳ thực thể nào khác trong CSDL (ví dụ: một Appointment hay Cooperation). Điều này cho phép tạo ra các "Thẻtương tác" (MessageType.INTERACTIVE\_CARD) đa năng và mạnh mẽ ngay trong giao diện chat.

#### Cụm Bất động sản

Đây là cụm cơ sở dữ liệu nền tảng, chịu trách nhiệm mô hình hóa một tài sản bất động sản với các thuộc tính vật lý, cố định và vị trí địa lý của nó. Một quyết định kiến trúc quan trọng là tách biệt hoàn toàn khái niệm "Bất động sản" (Property) khỏi "Tin đăng" (Listing), cho phép một tài sản có thể được đăng bán hoặc cho thuê nhiều lần mà không cần phải nhập lại các thông tin cốt lõi.

Ảnh có chứa văn bản, biểu đồ, ảnh chụp màn hình, Hình chữ nhật

Nội dung do AI tạo ra có thể không chính xác.

Giải thích các thực thể và mối quan hệ chính:

* ***Property (Bất động sản):***
  + Vai trò: Là thực thể trung tâm của cụm này, đại diện cho một tài sản vật lý cụ thể (ví dụ: một căn nhà, một mảnh đất). Nó liên kết trực tiếp đến chủ sở hữu (owner: User).
  + Các liên kết chính: Property tổng hợp các thông tin cơ bản và bất biến của một tài sản thông qua các mối quan hệ Nhiều-Một với các bảng định nghĩa khác.
* ***PropertyType, Direction, LegalStatus (Các bảng Định nghĩa):***
  + Vai trò: Đây là các bảng "danh mục", lưu trữ các lựa chọn có sẵn cho các thuộc tính của một bất động sản, ví dụ: "Căn hộ", "Nhà phố" (PropertyType); "Đông Nam", "Tây Bắc" (Direction); "Sổ hồng", "Hợp đồng mua bán" (LegalStatus).
  + Thiết kế: Mỗi bảng này đều chứa một trường code (ví dụ: APARTMENT, SOHONG) ngoài trường name. Quyết định này giúp phía frontend có thể dễ dàng ánh xạ các giá trị này với các icon hoặc logic hiển thị cụ thể mà không phụ thuộc vào chuỗi văn bản name có thể thay đổi.
* ***Location (Vị trí):***
  + Mối quan hệ: Một-Một với Property. Mỗi bất động sản chỉ có một vị trí duy nhất.
  + Vai trò: Mô hình hóa địa chỉ của tài sản một cách có cấu trúc.
  + Thiết kế nổi bật (Tích hợp GIS và Địa chỉ hành chính):
    - Trường point có kiểu dữ liệu PointField từ PostGIS, cho phép lưu trữ tọa độ kinh độ, vĩ độ một cách chính xác. Điều này là nền tảng cho các tính năng nâng cao như tìm kiếm theo bán kính, hiển thị trên bản đồ tương tác, hoặc phân tích khoảng cách.
    - Trường ward liên kết với model Ward từ thư viện vi\_address. Thông qua mối quan hệ cha-con đã được định nghĩa sẵn trong thư viện (Ward -> District -> City), hệ thống có thể dễ dàng truy xuất và hiển thị địa chỉ hành chính đầy đủ mà không cần phải lưu trữ trùng lặp.
* ***PropertyMedia (Media của Bất động sản):***
  + Mối quan hệ: Một-Nhiều với Property. Một bất động sản có thể có nhiều hình ảnh và video.
  + Vai trò: Lưu trữ thông tin về các tệp đa phương tiện.
  + Thiết kế nổi bật (Tích hợp Cloudinary): Thay vì lưu trữ trực tiếp file media trên server, trường url có kiểu CloudinaryField. Khi một tệp được tải lên, nó sẽ tự động được đẩy lên dịch vụ Cloudinary, và CSDL chỉ cần lưu lại đường dẫn URL do Cloudinary trả về. Trường public\_id cũng được lưu lại để có thể dễ dàng thực hiện các thao tác xóa hoặc cập nhật file trên Cloudinary sau này. Cách tiếp cận này giúp giảm tải đáng kể cho server và tối ưu hóa tốc độ phân phối nội dung đến người dùng cuối.

#### Cụm Tin đăng

Nếu cụm Bất động sản định nghĩa các thực thể "tĩnh", thì cụm Tin đăng mô hình hóa các hoạt động "động" của thị trường. Nó thể hiện một lần rao bán hoặc cho thuê cụ thể tại một thời điểm, đồng thời bao gồm toàn bộ hệ thống thương mại hóa (VIP, khuyến mãi) của nền tảng.

Ảnh có chứa văn bản, Giấy nhớ, biểu đồ, Kế hoạch

Nội dung do AI tạo ra có thể không chính xác.

**Giải thích các thực thể và mối quan hệ chính:**

* ***Listing (Tin đăng):***
  + Vai trò: Là thực thể trung tâm, đại diện cho một mẩu tin rao bán hoặc cho thuê. Nó liên kết với Property (tài sản vật lý) và User (người đăng tin) qua các mối quan hệ Nhiều-Một.
  + Các thuộc tính quan trọng: title, content (dạng RichTextField), price\_value, và các Enum trạng thái như Status (trạng thái giao dịch: Đang đăng, Đã bán...) và SpamCheckStatus (trạng thái kiểm duyệt tự động).
* ***BuySellDetail và RentalDetail (Thông tin chi tiết):***
  + Mối quan hệ: Một-Một với Listing.
  + Vai trò: Một quyết định thiết kế quan trọng là tách các trường thông tin chỉ thuộc về một loại tin đăng nhất định ra các bảng riêng. Ví dụ, deposit\_amount (tiền cọc) chỉ có ý nghĩa với tin cho thuê (RentalDetail), trong khi is\_mortgaged (đang thế chấp) chỉ liên quan đến tin mua bán (BuySellDetail). Cách làm này giúp cho bảng Listing chính luôn gọn gàng và tránh có quá nhiều trường bị NULL, tối ưu hóa cấu trúc CSDL.
* ***PropertyFeature và ListingPropertyFeatureValue (Đặc điểm Tin đăng):***
  + Vai trò: PropertyFeature là bảng định nghĩa các đặc điểm có thể có (ví dụ: "Số phòng ngủ", "Hướng ban công", "Có chỗ đỗ ô tô"). Nó được thiết kế linh hoạt với các Category và FeatureType khác nhau.
  + Mối quan hệ: ListingPropertyFeatureValue là bảng trung gian, hiện thực hóa mối quan hệ **Nhiều-Nhiều** giữa Listing và PropertyFeature.
  + Thiết kế nổi bật (Sử dụng JSONField): Thay vì tạo nhiều cột cho các kiểu dữ liệu khác nhau (text, number, boolean), trường value trong ListingPropertyFeatureValue có kiểu JSONField. Quyết định này mang lại sự linh hoạt tối đa, cho phép lưu trữ bất kỳ loại giá trị nào (số, chuỗi, boolean, hoặc thậm chí là một đối tượng JSON phức tạp) cho một đặc điểm mà không cần thay đổi cấu trúc bảng.
* ***VipType và ListingVip (Hệ thống VIP):***
  + Vai trò: VipType là bảng cấu hình, định nghĩa các gói VIP khác nhau (ví dụ: "VIP Kim Cương", "VIP Vàng") cùng với các thuộc tính kinh doanh như price\_per\_day (giá mỗi ngày) và sort\_priority (độ ưu tiên hiển thị).
  + Mối quan hệ: ListingVip là bảng ghi lại việc một Listing cụ thể đang ở trạng thái VIP nào. Mối quan hệ Một- Một giữa Listing và ListingVip đảm bảo mỗi tin đăng chỉ có một trạng thái VIP duy nhất tại một thời điểm. Trường end\_date cho biết thời điểm gói VIP hết hạn.
* ***PromotionRule (Quy tắc Khuyến mãi):***
  + Vai trò: Đây là bảng cấu hình cho các chương trình khuyến mãi, ví dụ: "Miễn phí tin đăng cho người mới" (FREE\_LISTING) hay "Giảm giá gói VIP" (DISCOUNT\_VIP).
  + Thiết kế linh hoạt: Bảng này cho phép định nghĩa các quy tắc phức tạp như trigger\_event (sự kiện kích hoạt, ví dụ: khi đăng ký) và validity\_duration (thời hạn hiệu lực), giúp hệ thống có thể tự động cấp phát khuyến mãi cho người dùng mà không cần sự can thiệp thủ công.

Nhìn chung, cụm CSDL này thể hiện rõ khả năng xử lý các nghiệp vụ phức tạp và đa dạng của Landify, từ việc mô tả chi tiết một bất động sản, quản lý trạng thái giao dịch, cho đến việc triển khai các mô hình kinh doanh và khuyến mãi một cách linh hoạt.

#### Cụm Tương tác Người dùng

Đây là cụm CSDL ghi lại tất cả các hành động và tương tác có ý nghĩa giữa người dùng và các tin đăng, hoặc giữa những người dùng với nhau. Các thực thể trong cụm này là kết quả trực tiếp của các hành vi trên ứng dụng, từ việc thể hiện sự quan tâm cho đến việc thực hiện các bước giao dịch cụ thể.

Ảnh có chứa văn bản, biểu đồ, hàng, Kế hoạch

Nội dung do AI tạo ra có thể không chính xác.

Giải thích các thực thể và mối quan hệ chính:

* ***Wishlist (Danh sách Yêu thích):***
  + Mối quan hệ: Bảng trung gian này hiện thực hóa mối quan hệ Nhiều-Nhiều giữa User và Listing.
  + Vai trò: Mỗi bản ghi trong Wishlist đại diện cho hành động một người dùng (user) đã "lưu" hoặc "thích" một tin đăng (listing) cụ thể. Điều này cho phép hệ thống xây dựng tính năng "Tin đã lưu" cho mỗi người dùng, đồng thời có thể thống kê được mức độ quan tâm của cộng đồng đối với mỗi tin đăng.
* ***Review (Đánh giá):***
  + Mối quan hệ: Một Review được tạo ra bởi một User và nhắm đến một Property cụ thể (chú ý là Property, không phải Listing, vì đánh giá là dành cho tài sản vật lý). Do đó, nó có hai mối quan hệ Nhiều-Một đến User và Property.
  + Vai trò: Lưu trữ các đánh giá của người dùng, bao gồm điểm số (rating) và nội dung bình luận (comment). Dữ liệu từ bảng này là nguồn đầu vào quan trọng để tính toán rating\_score trong UserProfile, giúp xây dựng uy tín cho người bán.
* ***Appointment (Lịch hẹn):***
  + Mối quan hệ: Một Appointment được tạo bởi một User (người hẹn) cho một Listing cụ thể (tin đăng muốn xem).
  + Vai trò: Đây là một thực thể nghiệp vụ quan trọng, mô hình hóa một lịch hẹn xem nhà chính thức. Nó chứa các thông tin cần thiết như thời gian hẹn (appointment\_date), ghi chú, và quan trọng nhất là status (trạng thái của lịch hẹn: Chờ xác nhận, Đã xác nhận, Đã hủy...). Bảng này là nền tảng cho tính năng quản lý lịch hẹn và gửi các thông báo nhắc nhở tự động.
* Cooperation (Hợp tác môi giới):
  + Mối quan hệ: Thực thể này liên kết phức tạp với ba đối tượng: Listing (tin đăng được đề nghị hợp tác), agent (người dùng gửi yêu cầu - môi giới), và owner (người dùng nhận yêu cầu - chủ tin đăng).
  + Vai trò: Cooperation mô hình hóa một quy trình nghiệp vụ chuyên biệt dành cho các nhà môi giới. Nó không chỉ là một tin nhắn thông thường mà là một "hợp đồng" đề nghị có trạng thái rõ ràng (status: Đang chờ, Chấp nhận, Từ chối...). Việc lưu trữ các bản ghi này cho phép hệ thống quản lý và theo dõi các mối quan hệ hợp tác giữa các thành viên trên nền tảng.

Về cốt lõi, các thực thể trong cụm tương tác này đóng vai trò then chốt trong việc biến Landify từ một trang chỉ để xem thông tin thành một nền tảng giao dịch sống động. Chúng ghi lại mọi hành vi quan trọng của người dùng, làm cơ sở để xây dựng các tính năng thông báo, thống kê, và tạo ra một môi trường tương tác có cấu trúc và hiệu quả.

#### Cụm Kiểm duyệt nội dung

Để xây dựng một cộng đồng an toàn và duy trì chất lượng thông tin trên nền tảng, một hệ thống kiểm duyệt nội dung mạnh mẽ và minh bạch là yêu cầu bắt buộc. Cụm CSDL này được thiết kế để mô hình hóa một quy trình kiểm duyệt khép kín, từ khi phát hiện vi phạm cho đến khi xử lý và giải quyết khiếu nại.

Ảnh có chứa văn bản, Giấy nhớ, chữ viết tay, ảnh chụp màn hình

Nội dung do AI tạo ra có thể không chính xác.

Giải thích các thực thể và mối quan hệ chính:

* ***Report (Báo cáo vi phạm):***
  + Mối quan hệ: Một Report được tạo bởi một User (người báo cáo - reporter).
  + Vai trò: Đây là điểm khởi đầu của quy trình kiểm duyệt, là thực thể ghi lại hành vi một người dùng báo cáo một nội dung nào đó là vi phạm. Nó chứa mô tả về vi phạm và trạng thái xử lý (status).
  + Thiết kế nổi bật (GenericForeignKey): Để hệ thống có thể báo cáo nhiều loại nội dung khác nhau (tin đăng, người dùng, bài viết, bình luận) mà không cần tạo nhiều bảng báo cáo riêng biệt, Report sử dụng một cặp trường reported\_item\_type (lưu loại đối tượng) và reported\_item\_id (lưu ID của đối tượng). Thiết kế này mang lại sự linh hoạt tối đa, cho phép dễ dàng mở rộng hệ thống kiểm duyệt cho bất kỳ loại nội dung mới nào trong tương lai.
* ***ModerationAction (Hành động Xử lý):***
  + Mối quan hệ: Một ModerationAction được thực hiện bởi một User có vai trò Admin (moderator). Nó có thể được kích hoạt bởi một Report (quan hệ Nhiều-Một, vì một báo cáo có thể dẫn đến nhiều hành động) hoặc được Admin thực hiện một cách chủ động (khi đó report có thể là NULL).
  + Vai trò: Thực thể này đóng vai trò là "nhật ký xử lý", ghi lại một cách minh bạch mọi hành động của đội ngũ quản trị viên. Nó lưu lại hành động cụ thể đã thực hiện (action\_type, ví dụ: Gỡ nội dung, Khóa tài khoản), lý do (reason), và đối tượng bị tác động (target\_object). Tương tự như Report, ModerationAction cũng sử dụng GenericForeignKey để có thể tác động lên nhiều loại đối tượng khác nhau.
* ***Protest (Kháng nghị):***
  + Mối quan hệ: Một Protest được tạo ra bởi một User (người kháng nghị - protester) và liên kết trực tiếp với một ModerationAction cụ thể thông qua mối quan hệ Một-Một.
  + Vai trò: Đây là bước cuối cùng trong quy trình, đảm bảo tính công bằng cho người dùng. Khi một người dùng cho rằng nội dung của họ bị xử lý sai, họ có thể tạo một Protest. Thực thể này sẽ lưu lại lý do kháng nghị và kết quả xử lý cuối cùng từ Admin (admin\_reviewer, resolution\_note, status).

***Luồng hoạt động tổng thể của quy trình:***

1. Một User tạo ra một Report nhắm vào một đối tượng (ví dụ: một Listing).
2. Admin xem xét Report và quyết định thực hiện một ModerationAction (ví dụ: gỡ Listing).
3. Chủ của Listing (là một User) không đồng ý và tạo ra một Protest liên kết với ModerationAction đó.
4. Một Admin khác (admin\_reviewer) xem xét Protest và đưa ra quyết định cuối cùng, cập nhật trạng thái của Protest.

Thiết kế này không chỉ giúp tự động hóa và quản lý các vi phạm một cách có hệ thống mà còn tạo ra một quy trình xử lý khiếu nại minh bạch, góp phần xây dựng một môi trường giao dịch công bằng và đáng tin cậy trên Landify.

### Thiết kế giao diện

## Kết quả đạt được

Dự án đã được hoàn thành và đưa vào vận hành với những chức năng trọng yếu, hình thành một nền tảng bất động sản đáng tin cậy. Bốn chức năng nổi bật nhất gồm có:

***Chức năng 1: Xác thực Danh tính Người dùng Toàn diện (eKYC)***

Đây là tính năng nền tảng giúp xây dựng sự tin cậy và minh bạch cho toàn bộ hệ thống.

*Ảnh có chứa văn bản, ảnh chụp màn hình, phần mềm, Phần mềm đa phương tiện

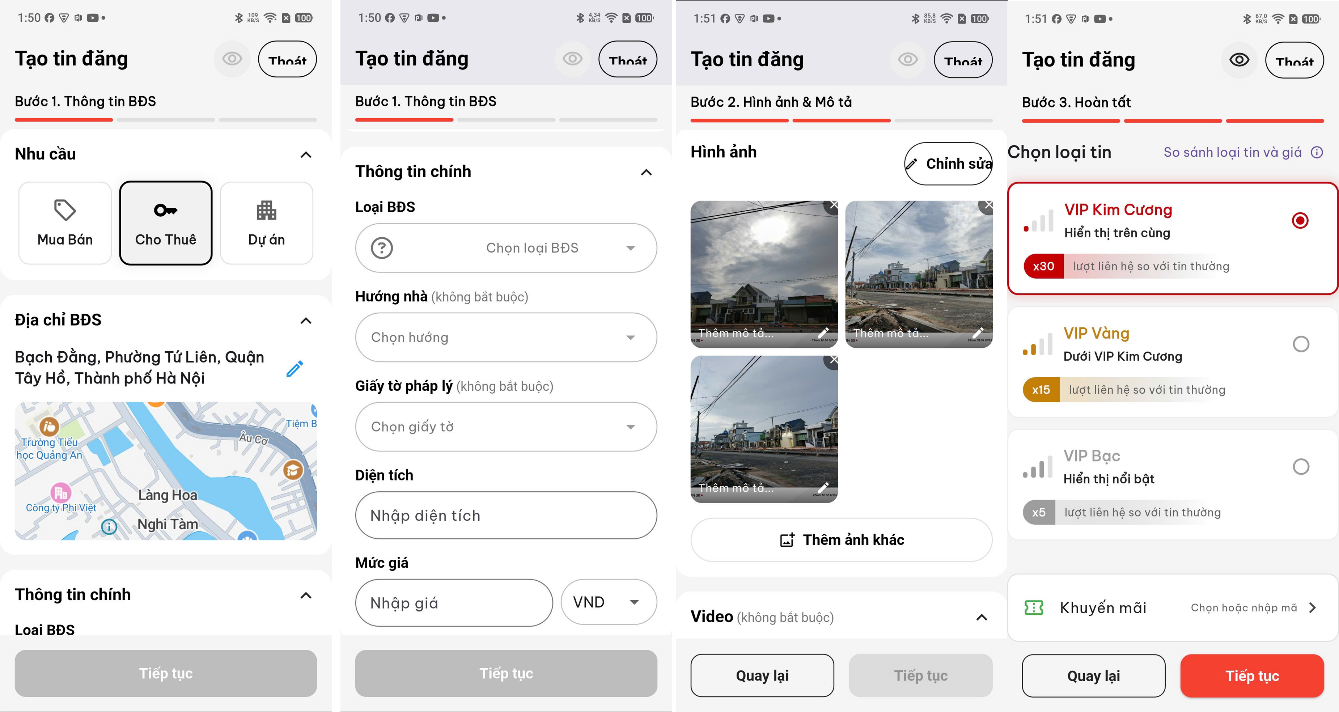
Nội dung do AI tạo ra có thể không chính xác.*

Hình . Chức năng: Xác thực danh tính

* **Quy trình sử dụng:**
  1. Người dùng truy cập vào trang cá nhân và chọn mục "Xác thực danh tính".
  2. Bước 1 - Quét CCCD: Ứng dụng mở giao diện camera. Người dùng đưa mặt trước của Căn cước công dân (CCCD) vào trong khung hình và chụp ảnh.
  3. Hệ thống tự động gửi ảnh đến server, sau đó gọi API của FPT.AI để trích xuất các thông tin như Họ tên, Ngày sinh, Quê quán... và cập nhật vào hồ sơ người dùng.
  4. Bước 2 - Xác thực khuôn mặt: Ứng dụng chuyển sang camera trước. Người dùng đưa khuôn mặt vào trong khung hình oval.
  5. Hệ thống tự động quay một video ngắn, sử dụng AI để kiểm tra đây là người thật (liveness check) và so sánh sự trùng khớp giữa khuôn mặt trong video với ảnh trên CCCD.
* Kết quả: Sau khi hoàn tất, tài khoản người dùng sẽ được gắn huy hiệu "Đã xác thực", tăng cường độ tin cậy khi thực hiện các giao dịch hoặc đăng tin trên nền tảng.

***Chức năng 2: Đăng đa dạng loại tin Bất động sản***

Chức năng cho phép người dùng, đặc biệt là các môi giới, cung cấp thông tin về bất động sản một cách chi tiết và đầy đủ nhất thông qua một quy trình 3 bước rõ ràng.

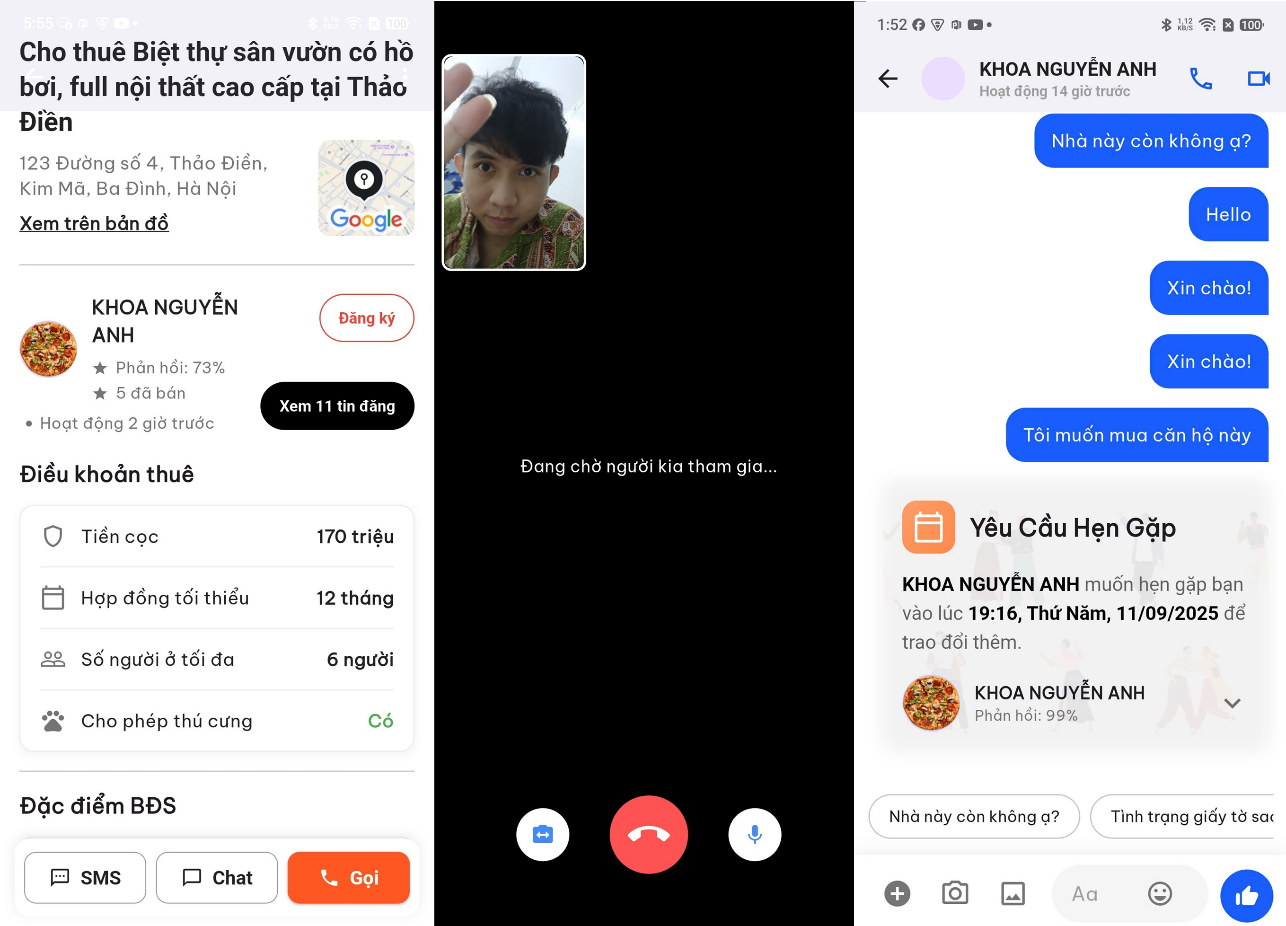


Hình . Chức năng: Tạo tin đăng

* Quy trình sử dụng:
  1. Người dùng chọn chức năng "Đăng tin" từ màn hình chính.
  2. Bước 1 - Thông tin BĐS: Người dùng nhập các thông tin cơ bản và bắt buộc như nhu cầu (Bán/Cho thuê), địa chỉ (hỗ trợ tìm kiếm và chọn trên bản đồ), loại hình BĐS, diện tích, mức giá, và các đặc điểm chi tiết (số phòng, hướng nhà, pháp lý...).
  3. Bước 2 - Hình ảnh & Mô tả: Người dùng tải lên nhiều hình ảnh (hỗ trợ chọn từ thư viện hoặc chụp mới), dán đường dẫn video Youtube/Tiktok, và soạn thảo mô tả chi tiết bằng trình soạn thảo văn bản giàu tính năng (rich text editor).
  4. Bước 3 - Hoàn tất: Người dùng lựa chọn loại tin đăng (Tin thường, VIP), áp dụng các mã khuyến mãi (nếu có), và tiến hành thanh toán để hoàn tất.
* Kết quả: Một tin đăng mới được tạo trên hệ thống với đầy đủ thông tin, hình ảnh trực quan. Ngay sau khi đăng, một tác vụ nền (Celery) sẽ tự động quét nội dung để kiểm tra spam, đảm bảo chất lượng cho toàn bộ nền tảng.

***Chức năng 3: Tương tác Trực tiếp qua Chat và Video Call***

Chức năng này phá bỏ rào cản giao tiếp, cho phép người mua và người bán kết nối tức thì, đẩy nhanh quá trình ra quyết định.



Hình . Chức năng: Chat & Video Call

* Quy trình sử dụng:
  1. Tại màn hình chi tiết một tin đăng, người dùng nhấn nút "Chat".
  2. Ứng dụng gọi API đến server Django để khởi tạo hoặc lấy thông tin phòng chat. Server trả về ID của phòng chat trên cả PostgreSQL và Firebase Firestore.
  3. Ứng dụng mở màn hình chat và bắt đầu lắng nghe (subscribe) vào document tương ứng trên Firestore để nhận/gửi tin nhắn theo thời gian thực.
  4. Trong màn hình chat, người dùng có thể nhấn vào biểu tượng Video Call.
  5. Ứng dụng tiếp tục gọi API Django để yêu cầu một token truy cập tạm thời cho dịch vụ Agora.
  6. Sau khi nhận được token, ứng dụng khởi tạo một phiên gọi video trực tiếp giữa hai người dùng thông qua Agora SDK.
* Kết quả: Người dùng có thể trao đổi thông tin, thương lượng và thậm chí là xem nhà từ xa một cách liền mạch và hiệu quả ngay trên ứng dụng.

# KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN

## Kết luận

### Những điều đã hoàn thành

### Những điều chưa hoàn thiện

Hiệu năng tải dữ liệu ban đầu: Mặc dù đã áp dụng các kỹ thuật tối ưu, việc tải đồng thời nhiều nguồn dữ liệu trên một số màn hình phức tạp (như trang chủ, trang chi tiết) đôi khi vẫn còn độ trễ, cần được tiếp tục tối ưu hóa.

Giao diện người dùng (UI/UX) trên một số thiết bị: Giao diện đã được xây dựng để đáp ứng đa dạng kích thước màn hình, tuy nhiên vẫn cần thêm các bước kiểm thử và tinh chỉnh để đảm bảo trải nghiệm hoàn hảo trên các thiết bị đặc thù (ví dụ: máy tính bảng, điện thoại màn hình gập).

Tính năng Chat và Gọi video còn ở mức cơ bản: Chức năng chat và gọi video đã hoạt động nhưng vẫn thiếu các tính năng nâng cao như gửi hình ảnh/file trong chat, chia sẻ màn hình, hay ghi âm cuộc gọi.

Hệ thống gợi ý và cá nhân hóa: Hệ thống hiện tại chủ yếu hiển thị dữ liệu dựa trên các bộ lọc thủ công của người dùng, chưa có cơ chế tự động gợi ý các tin đăng phù hợp dựa trên hành vi và lịch sử tìm kiếm.

## Hướng phát triển

Tích hợp hệ thống thanh toán trực tuyến: Tích hợp các cổng thanh toán như VNPay, MoMo để cho phép người dùng thanh toán các dịch vụ trực tiếp trên ứng dụng, ví dụ như mua các gói tin VIP, đặt cọc giữ chỗ bất động sản.

Ứng dụng Trí tuệ nhân tạo (AI) và Học máy (Machine Learning):

Xây dựng hệ thống gợi ý (Recommendation System) để tự động đề xuất các tin đăng phù hợp với sở thích và hành vi của người dùng.

Phát triển công cụ định giá bất động sản tự động dựa trên việc phân tích dữ liệu lịch sử và các đặc điểm của bất động sản.

Nâng cấp chatbot, tích hợp các mô hình ngôn ngữ lớn (LLM) để có thể trả lời các câu hỏi phức tạp hơn về pháp lý, quy hoạch.

TÀI LIỆU THAM KHẢO

(Theo chuẩn IEEE – *bỏ dòng này khi viết báo cáo*)

tối thiểu 40 trang, tôi đa 80 trang

PHỤ LỤC