

your\_project\_root/

├── frontend/ # Thư mục chứa code frontend (React, Vue, HTML/CSS/JS...)

│ └── ...

├── backend/ # Thư mục chứa code backend Python

│ ├── app/ # Gói (package) chính của ứng dụng backend

│ │ ├── \_\_init\_\_.py

│ │ ├── api/ # Chứa các file định nghĩa API endpoints (routes/views)

│ │ │ ├── \_\_init\_\_.py

│ │ │ ├── search\_api.py # <<< File định nghĩa endpoint /api/search

│ │ │ └── ... (các API khác)

│ │ ├── services/ # Chứa business logic, nơi điều phối các tác vụ

│ │ │ ├── \_\_init\_\_.py

│ │ │ ├── search\_service.py # <<< Hàm chính điều phối tìm kiếm nằm ở đây

│ │ │ └── ... (các service khác)

│ │ ├── search\_logic/ # (HOẶC TÊN KHÁC NHƯ 'features/search', 'modules/search'...)

│ │ │ ├── \_\_init\_\_.py

│ │ │ ├── preprocessor.py # <<< Chứa hàm tiền xử lý, từ đồng nghĩa, undertext...

│ │ │ ├── tfidf\_engine.py # <<< Chứa hàm/lớp tính điểm TF-IDF, load model

│ │ │ ├── tag\_engine.py # <<< Chứa hàm/lớp tính điểm khớp Tag, tương tác

│ │ │ ├── ranker.py # <<< Chứa hàm kết hợp điểm số và xếp hạng

│ │ │ └── utils.py # (Tùy chọn) Các hàm tiện ích dùng chung cho search

│ │ ├── models/ # (Nếu dùng ORM) Chứa định nghĩa các model CSDL

│ │ │ ├── \_\_init\_\_.py

│ │ │ ├── dia\_danh.py

│ │ │ ├── tag.py

│ │ │ └── dia\_danh\_tag.py

│ │ ├── core/ # Các thành phần cốt lõi (config, db connection...)

│ │ │ ├── \_\_init\_\_.py

│ │ │ ├── config.py # <<< Có thể chứa cấu hình trọng số, đường dẫn model...

│ │ │ └── database.py # <<< Thiết lập kết nối CSDL

│ │ ├── main.py # File khởi tạo ứng dụng chính (ví dụ Flask app)

│ │ └── ... (các module khác)

│ ├── data/ # Thư mục lưu trữ các file dữ liệu không phải code

│ │ ├── models/ # Chứa các model đã huấn luyện

│ │ │ ├── tfidf\_model.pkl

│ │ │ └── tfidf\_vectorizer.pkl

│ │ ├── dictionaries/

│ │ │ └── synonyms.json # File từ điển đồng nghĩa

│ │ └── ...

│ ├── tests/ # Thư mục chứa các bài kiểm thử (unit tests, integration tests...)

│ │ ├── test\_search/

│ │ ├── test\_preprocessing.py

│ │ ├── test\_ranking.py

│ │ └── ...

│ ├── venv/ # Thư mục môi trường ảo Python

│ ├── requirements.txt # File liệt kê các thư viện cần cài đặt

│ ├── run.py # File để chạy server (ví dụ: chạy Flask/Django dev server)

│ ├── .env # File chứa biến môi trường (API keys, DB credentials...)

├── README.md

├── .gitignore

└── ...

**1. Vị trí cụ thể của các hàm tìm kiếm trong file .py (theo cấu trúc ví dụ):**

Dựa trên quy trình xử lý và cấu trúc thư mục tôi đã gợi ý ở trên, đây là nơi các "hàm" hoặc "logic" tương ứng thường sẽ được đặt:

* **Hàm điều phối chính** (tương đương handle\_search\_request mà tôi mô tả): Sẽ nằm trong file **backend/app/services/search\_service.py**. Hàm này nhận query từ API và gọi các hàm/lớp xử lý con.
* **Logic tiền xử lý query** (từ đồng nghĩa, tách từ underthesea, trích xuất keyword/tag): Sẽ nằm trong file **backend/app/search\_logic/preprocessor.py**.
* **Logic tính điểm TF-IDF:** Sẽ nằm trong file **backend/app/search\_logic/tfidf\_engine.py**. File này cũng có thể chứa logic load model TF-IDF và vectorizer.
* **Logic tính điểm khớp Tag:** Sẽ nằm trong file **backend/app/search\_logic/tag\_engine.py**. File này có thể chứa logic truy vấn CSDL để lấy tag của địa điểm.
* **Logic kết hợp điểm số và xếp hạng:** Sẽ nằm trong file **backend/app/search\_logic/ranker.py**.
* **Hàm xử lý yêu cầu HTTP (API endpoint):** Sẽ nằm trong file **backend/app/api/search\_api.py**. Hàm này chỉ đơn giản là nhận request, gọi hàm trong search\_service.py, và trả về response.

**Tóm lại:** Các hàm thực hiện các bước tính toán và xử lý cụ thể (tiền xử lý, tính TF-IDF, tính Tag, xếp hạng) nằm trong các file riêng biệt trong thư mục search\_logic/. Hàm điều phối tổng thể nằm ở lớp services/. Hàm tiếp nhận yêu cầu web nằm ở lớp api/.

**2. Cấu trúc file có "nhiều quá không" cho website 3 trang?**

Tôi hiểu cảm giác của bạn rằng cấu trúc gợi ý có vẻ hơi nhiều lớp và file so với một website chỉ có 3 trang chính. Tuy nhiên, có một vài lý do quan trọng để cân nhắc áp dụng cấu trúc có tổ chức ngay từ đầu, ngay cả với dự án nhỏ/vừa:

* **Bản chất của chức năng tìm kiếm:** Mặc dù website chỉ có 3 trang, nhưng bản thân chức năng tìm kiếm bạn đang xây dựng (kết hợp TF-IDF, Tags, Từ đồng nghĩa) đã bao gồm nhiều bước logic phức tạp và riêng biệt. Việc chia nhỏ giúp quản lý chính sự phức tạp *nội tại* của chức năng này.
* **Dễ bảo trì và sửa lỗi:** Khi bạn cần tinh chỉnh thuật toán TF-IDF, bạn biết chính xác cần mở file tfidf\_engine.py. Khi cần sửa lỗi lấy tag từ CSDL, bạn vào tag\_engine.py. Nếu tất cả nằm trong một file, việc tìm và sửa sẽ khó khăn hơn nhiều, dễ gây ảnh hưởng chéo.
* **Khả năng mở rộng (Scalability):** Website của bạn *hiện tại* có thể đơn giản, nhưng trong tương lai bạn có thể muốn thêm tính năng (ví dụ: tìm kiếm theo khoảng giá, lọc theo đánh giá người dùng, gợi ý địa điểm tương tự...). Một cấu trúc có sẵn các module tách biệt sẽ giúp việc thêm các tính năng mới dễ dàng hơn rất nhiều mà không cần cấu trúc lại toàn bộ code.
* **Dễ đọc và làm việc nhóm:** Ngay cả khi chỉ một mình bạn làm, sau một thời gian quay lại dự án, việc đọc code được tổ chức rõ ràng sẽ dễ hơn. Nếu có người khác tham gia, cấu trúc chuẩn sẽ giúp họ nhanh chóng nắm bắt luồng xử lý.
* **Kiểm thử (Testing):** Chia nhỏ thành các file/hàm giúp bạn viết unit test cho từng phần dễ dàng hơn, đảm bảo chất lượng code.

**Bắt đầu:** Người dùng nhập văn bản vào ô tìm kiếm trên trình duyệt.

**Bước 1: Tiếp nhận yêu cầu (Frontend -> Backend API)**

1. **Input:** Văn bản truy vấn của người dùng (VD: "biển đẹp vắng người").
2. **File xử lý:** backend/app/api/search\_api.py (Ví dụ: hàm xử lý route /api/v1/search).
3. **Hành động:**
   * Nhận yêu cầu HTTP POST hoặc GET chứa văn bản truy vấn.
   * Trích xuất văn bản truy vấn.
   * Gọi hàm xử lý logic chính trong service layer, truyền văn bản truy vấn vào.
4. **Output (cho bước tiếp theo):** Gọi hàm search\_service.find\_relevant\_locations("biển đẹp vắng người").
5. **Dữ liệu đi tiếp đến:** backend/app/services/search\_service.py.

**Bước 2: Điều phối & Tiền xử lý truy vấn**

1. **Input:** Văn bản truy vấn thô ("biển đẹp vắng người").
2. **File xử lý:** backend/app/services/search\_service.py (Ví dụ: hàm find\_relevant\_locations).
3. **Hành động:**
   * Gọi hàm tiền xử lý trong module preprocessor.
4. **File được gọi:** backend/app/search\_logic/preprocessor.py (Ví dụ: hàm preprocess\_and\_expand).
   * **Hành động bên trong preprocessor.py:**
     + Nhận query thô.
     + Áp dụng từ đồng nghĩa (VD: "biển" -> "biển", "bãi biển").
     + Dùng underthesea tách từ, chuẩn hóa (lowercase, bỏ dấu câu...).
     + Trích xuất các keyword/tag tiềm năng (VD: ['biển', 'bãi biển', 'đẹp', 'vắng']).
   * **Kết quả trả ra từ preprocessor.py:** Một cấu trúc dữ liệu chứa các thành phần đã xử lý, ví dụ: {'tokens\_for\_tfidf': ['biển', 'đẹp', 'vắng', 'người', 'bãi\_biển'], 'extracted\_tags': ['biển', 'bãi biển', 'đẹp', 'vắng']}.
5. **Output (trong search\_service.py):** Biến chứa kết quả tiền xử lý (ví dụ: processed\_data).
6. **Dữ liệu đi tiếp:** processed\_data sẽ được dùng để gọi các hàm tính điểm.

**Bước 3: Tính điểm TF-IDF (Song song)**

1. **Input:** Danh sách token đã xử lý từ processed\_data['tokens\_for\_tfidf'].
2. **File xử lý (được gọi bởi search\_service.py):** backend/app/search\_logic/tfidf\_engine.py (Ví dụ: hàm calculate\_scores).
3. **Hành động:**
   * Nhận token truy vấn.
   * Vector hóa token truy vấn bằng TF-IDF Vectorizer đã load.
   * Tính toán độ tương đồng cosine giữa vector truy vấn và vector của *từng* mô tả địa điểm (đã được tính toán và lưu trữ trước hoặc tính toán nhanh).
4. **Kết quả trả ra từ tfidf\_engine.py:** Một dictionary chứa điểm TF-IDF cho mỗi địa điểm, ví dụ: {'DD001': 0.6, 'DD004': 0.1, 'DD511': 0.85, ...}.
5. **Dữ liệu đi tiếp đến:** Kết quả này (tfidf\_scores) được trả về cho search\_service.py.

**Bước 4: Tính điểm Khớp Tag (Song song)**

1. **Input:** Danh sách keyword/tag đã trích xuất từ processed\_data['extracted\_tags'].
2. **File xử lý (được gọi bởi search\_service.py):** backend/app/search\_logic/tag\_engine.py (Ví dụ: hàm calculate\_scores).
3. **Hành động:**
   * Nhận các tag/keyword từ truy vấn.
   * Đối với mỗi địa điểm, lấy danh sách các tag đã gán cho nó (có thể truy vấn CSDL hoặc từ cache).
   * So sánh và tính điểm khớp giữa tag truy vấn và tag địa điểm.
4. **Kết quả trả ra từ tag\_engine.py:** Một dictionary chứa điểm khớp tag cho mỗi địa điểm, ví dụ: {'DD001': 0.9, 'DD004': 0.2, 'DD511': 0.3, ...}.
5. **Dữ liệu đi tiếp đến:** Kết quả này (tag\_scores) được trả về cho search\_service.py.

**Bước 5: Kết hợp điểm và Xếp hạng**

1. **Input (trong search\_service.py):** tfidf\_scores (từ Bước 3) và tag\_scores (từ Bước 4).
2. **File xử lý (được gọi bởi search\_service.py):** backend/app/search\_logic/ranker.py (Ví dụ: hàm combine\_and\_rank).
3. **Hành động:**
   * Nhận hai bộ điểm.
   * Lấy các trọng số kết hợp (từ config.py chẳng hạn).
   * Với mỗi địa điểm có trong cả hai bộ điểm (hoặc xử lý trường hợp chỉ có ở một bộ), tính điểm tổng hợp: final\_score = (w1 \* tfidf\_score) + (w2 \* tag\_score).
   * Sắp xếp các địa điểm theo final\_score giảm dần.
4. **Kết quả trả ra từ ranker.py:** Một danh sách đã sắp xếp chứa ID của các địa điểm phù hợp nhất, ví dụ: ['DD001', 'DD511', 'DD004', ...].
5. **Dữ liệu đi tiếp đến:** Danh sách ID đã sắp xếp (ranked\_ids) được trả về cho search\_service.py.

**Bước 6: Chuẩn bị dữ liệu trả về**

1. **Input (trong search\_service.py):** Danh sách ID địa điểm đã sắp xếp (ranked\_ids).
2. **File xử lý:** backend/app/services/search\_service.py.
3. **Hành động:**
   * Lấy top N ID từ danh sách (ví dụ: 10 ID đầu tiên).
   * Truy vấn CSDL (có thể gọi hàm trong module data access hoặc dùng ORM trong models/) để lấy thông tin chi tiết (tên, ảnh, mô tả ngắn...) cho các ID này.
   * Định dạng dữ liệu thành một cấu trúc phù hợp để trả về cho API (ví dụ: list các dictionary).
4. **Output (cho bước tiếp theo):** Dữ liệu địa điểm đã được định dạng (ví dụ: results\_data).
5. **Dữ liệu đi tiếp đến:** results\_data được trả về cho backend/app/api/search\_api.py.

**Bước 7: Trả về Response**

1. **Input:** Dữ liệu địa điểm đã được định dạng (results\_data).
2. **File xử lý:** backend/app/api/search\_api.py.
3. **Hành động:**
   * Tạo một HTTP Response thành công (status code 200).
   * Đặt results\_data vào phần body của response, thường là dưới dạng JSON.
   * Gửi response về cho trình duyệt/frontend.

**Kết thúc:** Frontend nhận dữ liệu JSON và hiển thị danh sách kết quả tìm kiếm cho người dùng.

Sơ đồ này mô tả luồng dữ liệu tuần tự và song song qua các module/file khác nhau, mỗi file chịu trách nhiệm một phần cụ thể, giúp hệ thống có tổ chức và dễ quản lý hơn.