# Medicine Product Search and Scraping Tool

## Project Overview

This project is a comprehensive web application for searching and scraping medicine product information from multiple medical databases and websites. It consists of two main components:

1. A Django backend API service for web scraping and data management

2. A Flask frontend UI for user interaction

## System Architecture

### Backend (Django)

Located in `/scrapper/med\_scrapper/`:

- Django REST API service

- Handles web scraping from multiple sources (MHRA, BNF, EMC)

- Manages database operations and data storage

- Processes search queries and results

Key Components:

- `core/models.py`: Database models for storing websites, search queries, results, and derived products

- `core/views.py`: API endpoints and business logic

- `core/webscrapper.py`: Web scraping functionality

- `core/serializers.py`: Data serialization for API responses

### Frontend (Flask)

Located in `/UI/`:

- Flask web application

- Provides user interface for searching and viewing results

- Communicates with Django backend via REST API

Key Components:

- `app.py`: Flask application logic and routes

- `templates/index.html`: Main user interface template

## Features

1. \*\*Multi-Source Search\*\*

- Searches across multiple medical databases (MHRA, BNF, EMC)

- Displays results in parallel columns for easy comparison

2. \*\*Result Filtering\*\*

- Real-time client-side filtering of search results

- Each source's results can be filtered independently

3. \*\*Manual Corrections\*\*

- Users can provide manual corrections when search results are inadequate

- Corrections are stored and tracked in the database

4. \*\*Derived Product Information\*\*

- System extracts structured information from search results

- Captures API, strength, and dosage form information

- Links derived information to source results or manual corrections

## Database Schema

### Core Models

1. \*\*Website\*\*

- name: Name of the source website

- url: Base URL of the website

- created\_at: Timestamp

2. \*\*SearchQuery\*\*

- query\_text: User's search query

- timestamp: When the search was performed

3. \*\*SearchResult\*\*

- search\_query: Reference to SearchQuery

- website: Reference to Website

- title: Product name/title

- position: Result position in search

- product\_url: URL to product page

- raw\_data: Additional scraped data

- is\_user\_selected: Whether user selected this result

4. \*\*ManualCorrection\*\*

- search\_query: Reference to SearchQuery

- corrected\_text: User's manual correction

- correction\_timestamp: When correction was made

5. \*\*DerivedProductInstance\*\*

- search\_query: Reference to SearchQuery

- source\_result: Reference to SearchResult (if derived from a result)

- Additional fields for structured data (API, strength, dosage form)

\*\*Configuration\*\*

- Ensure Django server is running on port 8001

- Configure `DJANGO\_API\_BASE\_URL` in Flask's `app.py` if needed

- Set up website entries in Django admin for each source

## API Endpoints

1. \*\*Search API\*\*

- POST `/api/search/`: Perform new search

- GET `/api/search\_queries/{id}/`: Get search query details

2. \*\*Feedback API\*\*

- POST `/api/feedback/`: Submit user feedback

- GET `/api/derived\_products/{search\_id}/`: Get derived products

## Technical Requirements

- Python 3.x

- Django 4.x

- Django REST Framework

- Flask

- SQLite database (default)

- Modern web browser with JavaScript enabled

## Development Notes

1. \*\*Adding New Sources\*\*

- Add new website entry in Django admin

- Create scraper function in `webscrapper.py`

- Update `WEBSITES` list in Flask app

- Update UI template to handle new source

## Security Considerations

- Input sanitization on both frontend and backend

- CORS properly configured for API access

- SQL injection prevention through ORM

- XSS protection in templates