Name-Recipe Project

Overview

This project contains two components required by the assignment:

1. **Name Matching** — a small API that finds the closest matching name(s) from a local list and returns similarity scores.
2. **Recipe Chatbot (Retrieval + Template)** — an API that accepts a list of ingredients and returns matching recipes (structured results) and a conversational suggestion.

The project is implemented using **FastAPI** for the backend, **RapidFuzz** for name similarity, and a small JSON dataset for recipes. A minimal static web UI is included to demo both features.

Files & Structure

```
name-recipe-project/
├─ README.md
├─ requirements.txt
├─ data/
│  ├─ names_list.txt
│  └─ recipes.json
├─ app/
│  ├─ name_matcher.py
│  ├─ recipe_engine.py
│  └─ main.py
├─ ui/
│  ├─ index.html
│  └─ app.js
└─ sample_runs.md
```

Prerequisites

- Python 3.9 or newer
- pip (Python package installer)
- Optional: Git Bash or PowerShell on Windows (instructions below)

Setup Instructions (step-by-step)

1. Clone or copy the project to your machine

Place the `name-recipe-project` folder on your Desktop or any working directory.

2. Create a Python virtual environment

Open the terminal of your choice and run:

Windows (PowerShell)

```
cd "C:\Users\<YourUser>\Desktop\name-recipe-project"
python -m venv .venv
```

Git Bash / WSL / macOS / Linux

```
cd ~/Desktop/name-recipe-project
python -m venv .venv
```

3. Activate the virtual environment

PowerShell (recommended on Windows)

```
# If you see an ExecutionPolicy error, run the command below AS ADMIN once:
# Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser
.venv\Scripts\Activate.ps1
```

Command Prompt

```
.venv\Scripts\activate.bat
```

Git Bash

```
source .venv/Scripts/activate
```

You should now see the virtual environment prefix in your prompt, for example `(.venv)`.

4. Install dependencies

With the virtual environment active:

```
pip install --upgrade pip
pip install -r requirements.txt
```

If you don't have a `requirements.txt`, install the required packages manually:

```
pip install fastapi uvicorn[standard] rapidfuzz pydantic python-multipart
jinja2 aiofiles
```

Why PowerShell vs Git Bash on Windows

- **PowerShell** is the native Windows shell with full support for `.ps1` scripts. When you create a Python venv on Windows, the activation script (`Activate.ps1`) is a PowerShell script. Running that script in PowerShell is the standard way to activate the venv.
- **Git Bash** emulates a Unix-like environment and uses a different activation path (`source .venv/Scripts/activate`). Both shells work — use whichever you're comfortable with.
- If you get `ExecutionPolicy` errors in PowerShell, this is a security policy preventing local script execution. Running `Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser` (as an administrator) allows running local scripts for your user safely.

Short phrase : "I used a Python virtual environment (venv). On Windows I activated it with PowerShell because Windows provides a PowerShell activation script for venv; Git Bash also works with a different command. I made sure dependencies were installed within the venv to keep the environment isolated."

How to run the project

With virtual environment active:

```
uvicorn app.main:app --reload --host 127.0.0.1 --port 8000
```

Open a browser and go to:

- Project UI (static files): `http://127.0.0.1:8000`
- API docs (Swagger UI): `http://127.0.0.1:8000/docs`

API Endpoints (for verification)

POST `/match_name`

Request JSON:

```
{ "query": "Gita", "top_k": 5 }
```

Response example:

```
{
  "best_match": { "name": "Gita", "score": 100 },
```

```
  "ranked": [
    {"name":"Gita","score":100},
    {"name":"Geeta","score":88}
  ]
}
```

POST `/recipe_chat`

Request JSON:

```
{ "ingredients": ["egg", "onion"], "top_k": 3 }
```

Response example:

```
{
  "matches": [
    {"id":1, "title":"Egg & Onion Scramble", "score":2, "ingredients":
["egg","onion"], "instructions":"..."}
  ],
  "response_text":
"Try 'Egg & Onion Scramble' (match score 2):\nIngredients: egg, onion..."
}
```

Sample input / expected output (copy to `sample_runs.md`)

Name match test Input: `{"query":"Geetha"}` Expected: best match should be a similar name from `data/names_list.txt` with a high similarity score.

Recipe test Input: `{"ingredients":["egg","onion"]}` Expected: the response contains the recipe(s) that contain `egg` and/or `onion` with `response_text` presenting a conversational suggestion.

Explain work (step-by-step talking points)

Use this script during the interview — concise and clear.

1. **Problem statement** — "The task required building two components: a name-matcher that returns best and ranked name matches with similarity scores, and a recipe chatbot that suggests recipes based on input ingredients."

2. **Data collection** — "I prepared two small datasets: a `names_list.txt` (≥30 names) for name-matching and `recipes.json` (sample of recipes with `title`, `ingredients`, `instructions`). The recipes were either sourced from public datasets/synthetic examples and normalized for ingredient tokens."

3. **Approach & design choices** — "I prioritized a solution that is easy to run on a standard laptop:

4. For name matching I used `rapidfuzz` (token-based similarity) for speed and accuracy on small lists.

5. For the recipe chatbot I used a retrieval-based approach: compute ingredient overlap and return the best matches and a templated friendly response. This avoids heavy local model dependencies and still produces useful outputs."

6. **Implementation details** — "Backend: FastAPI exposing two endpoints (`/match_name`, `/recipe_chat`). Name matching logic is in `app/name_matcher.py` and recipe logic in `app/recipe_engine.py`. There's a tiny static UI to test quickly."

7. **Why these tools** — "RapidFuzz gives high-quality string similarity fast. FastAPI is lightweight and fast to develop and provides interactive API docs (`/docs`). Using a simple retrieval approach ensures the system is deterministic and easy to evaluate."

8. **Demonstration plan** — "I will start the server with `uvicorn`, open `/docs`, and run two example calls: one to `/match_name` and one to `/recipe_chat`. Then I'll show the static UI to demonstrate the user-facing side."

9. **Limitations & future improvements** — "I would add: larger recipe dataset, ingredient normalization (units and synonyms), fuzzy ingredient matching using embeddings (sentence-transformers + FAISS) for better recall, and optionally a small generator for nicer conversational replies (RAG) if compute is available."

10. **Testing & verification** — "I included `sample_runs.md` and example JSONs in the README for the interviewer to validate the results quickly."

11. **Closing** — "This solution balances reproducibility, speed, and clarity: it is easy to run locally, easy for the interviewer to validate, and extensible for production improvements."
