Installation and Running (via pip)

Prerequisites

- Python 3.x
- PostgreSQL running locally or reachable over the network
- Environment variables set for DB connection (host, dbname, user, password, port)

Create and activate a virtual environment

```
python -m venv .venv
# macOS/Linux
source .venv/bin/activate
# Windows
.venv\Scripts\activate
```

Install dependencies and the package

```
python -m pip install --upgrade pip
pip install -r requirements.txt
pip install -e .
```

Run the application

```
python src/app.py
```

Run tests and coverage

```
pytest
pytest --cov-report=term-missing
```

Run linting (fix for CI “pylint not found”)

python -m pip install pylint in case there is some completely unforeseen issue with your lint installation but it should work without the pip install after installing the requirements.

Place in terminal to lint src/ :

```
python -m pylint --fail-under=10 src/app.py src/board/*.py
```

Installation and Running (via uv)

Prerequisites

Install uv (one-time):

pip install uv

Create an isolated environment and install dependencies

```
uv venv
# macOS/Linux
source .venv/bin/activate
# Windows
.venv\Scripts\activate

uv pip install -r requirements.txt
uv pip install -e .
```

Run the application using uv

```
uv run flask --app src/app.py run
```

Run tests with uv

```
uv run pytest
uv run pytest --cov-report=term-missing
```

Dependency Graph Summary

The dependency graph shows the Flask web layer (`app.py`) orchestrating an ETL flow: scrape → clean → load → query/analyze. The scraper module depends on HTTP + HTML parsing libraries (urllib3 and BeautifulSoup) to retrieve and parse GradCafe pages. The cleaning module performs normalization and parsing of decision outcomes, dates, and optional numeric fields before persistence. The loader module is responsible for database connectivity and inserting sanitized records using psycopg. The analysis/query module performs read-only, parameterized SQL queries and enforces a strict LIMIT clamp to prevent large or unbounded reads. Overall, the graph reflects a clean separation of concerns: web routes call orchestrators, which call domain modules that each focus on one pipeline stage.

SQL Injection Defenses (what changed and why it's safe)

What changed

The code avoids building SQL strings by concatenating user input and instead uses parameterized queries and psycopg SQL composition. Values like term names, “international” flags, universities, programs and more are passed as parameters using placeholders, so the database driver treats them strictly as data, not executable SQL. This works as an effective counter against SQL injectors from bad apples. For dynamic identifiers the code uses `psycopg.sql.Identifier` rather than interpolating raw strings. In the screenshot included in the module there is a mention of version vulnerabilities for installed packages and to minimize them I updated the requirements.txt to versions that lack those weaknesses where available.

Why it’s safe

Parameterized queries ensure user controlled text never changes the query structure. Identifier composition prevents attackers from injecting malicious SQL via table/column names, because identifiers are safely quoted/escaped by the driver. Additionally, the analytics layer clamps `LIMIT` to a safe range (1–100), reducing the risk of resource exhaustion even if someone tries to request huge result sets. Together, these measures protect both integrity (no injected writes) and confidentiality while keeping queries predictable.

Least-Privilege DB Configuration (permissions and why)

Principle

The application should connect to PostgreSQL using a dedicated role with only the permissions it needs and nothing more. This minimizes the blow out if any sort of breach happens as the room for power is virtually nil.

Recommended roles

1. App runtime role (User permissions)
    - Permissions: `CONNECT` to the database, `USAGE` on the schema, and only `SELECT/INSERT/UPDATE` on the application table(s) it uses (e.g., `applicants`).

    - Avoid granting: `SUPERUSER`, `CREATEDB`, `CREATEROLE`, unrestricted `DROP`, or access to other schemas/tables.

2. Migration/admin role (which I only used during setup)
    - Permissions to `CREATE TABLE`, `ALTER`, `DROP` during schema changes.

- Not used by the running web app.

SQL Setup

```
-- 1) Database access
GRANT CONNECT ON DATABASE sipoftea_db TO sipoftea_db_user;

-- 2) Schema permissions needed for CREATE TABLE
GRANT USAGE, CREATE ON SCHEMA public TO sipoftea_db_user;

-- 3) Optional: if table already exists and owned by someone else
GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE public.applicants TO
sipoftea_db_user;

-- 4) Sequence rights for SERIAL/IDENTITY IDs (if table pre-exists)
GRANT USAGE, SELECT ON SEQUENCE public.applicants_p_id_seq TO sipoftea_db_user;
```