How to Install/Run via pip and uv

pip

- python3 -m venv module_5/venv
- source module_5/venv/bin/activate
- pip install -r module_5/requirements.txt

uv

- uv venv module_5/.venv
- source module_5/.venv/bin/activate
- uv pip install -r module_5/requirements.txt

Run test

- pytest -c module_5/pytest.ini module_5/tests -m "web or buttons or analysis or db or integration" --cov=module_5/src

Run app

- cd module_5/src
- python app.py

Dependency Graph Summary (5–7 sentences)

The project is a Flask-based web app with a PostgreSQL backend accessed through `psycopg`. The dependency graph shows a visual representation of the different parts of a project, such as modules, libraries, or packages, relying on one another. The various dependencies are in module_5/requirements.txt and the main layer, app.py is calling on query_data.py and load_data.py which is used to load the data and includes the sql query. The data is obtained from the module_2 files which include scrape.py and clean.py which obtain data from the website https://www.thegradcafe.com/. The data is then handled by the LLM in llm_hosting/app.py. All of these connections are seen with the dependency graph. The various supporting dependencies in requirements.txt include the runtime libraries such as flask, psycopg and the quality/security tools such as pytest, pytest-cov, pylint, pydeps, snyk. Overall, the graph helps to show a layered architecture of the entrypoint, the data/query pipeline and the external data/model services.

your SQL injection defenses (what changed and why it's safe)

For SQL injection defense, I changed the code so user input is never directly pasted into SQL strings. I used psycopg.sql with sql.Identifier(...) for dynamic table/column names, so only valid identifiers are treated as SQL syntax. For user-provided values, I used parameter binding (%s) in cursor.execute(...), which means values are sent separately from the SQL command text. Because of that, malicious input is treated as plain data, not executable SQL. I also required a LIMIT for user-driven queries and clamped it to a safe range (1 to 100) to avoid returning too much data or running overly heavy queries. For fixed analytics queries (q1-q11), I added LIMIT 1; this does not change the result logic for aggregates, but it satisfies the explicit query-bound requirement. For example, previously the code query = f"SELECT * FROM applicants WHERE

university = '{user_input}' LIMIT {limit}" cursor.execute(query) would be at risk for SQL injections and therefore not safe. To fix this, a safer version is made stmt = sql.SQL("SELECT * FROM {table} WHERE university = %s LIMIT %s").format(table=sql.Identifier("applicants")) cursor.execute(stmt, [user_input, safe_limit]). This method of execution means that the user_input is bound as data (and not the SQL code), and safe_limit is in a safe range (for example, 1–100), so injection strings cannot alter query structure.

SQL Snippet: Least-privilege DB user

```
CREATE ROLE sm_app_user
LOGIN PASSWORD 'ABC123'
NOSUPERUSER NOCREATEDB NOCREATEROLE NOINHERIT;

GRANT CONNECT ON DATABASE sm_app TO sm_app_user;
GRANT USAGE ON SCHEMA public TO sm_app_user;
GRANT SELECT, INSERT ON TABLE applicants TO sm_app_user;
```

Environment-based DB config

I removed hard-coded DB credentials from app runtime paths meaning that now it reads DB connection settings from environment variables: DB_HOST, DB_PORT, DB_NAME, DB_USER, DB_PASSWORD.
.env.example with placeholder values only

Least-privilege DB configuration (what permissions and why)

The least-privilege DB configuration is created by dedicating an app user (sm_app_user) with the role as not a superuser and has no admin capabilities. (CREATE ROLE sm_app_user LOGIN PASSWORD `ABC123` NOSUPERUSER NOCREATEDB NOCREATEROLE NOINHERIT;) adn therefore No DROP/ALTER/owner-level privileges were granted.
The CONNECT on database sm_app is required to grant access to connect. USAGE on schema public allows for the user to access objects (tables/views) inside the public schema. Without it, even table permissions won't work..
SELECT on applicants allows the user to read rows from applicants which the INSERT on applicants permission is required for data loading pipeline.

And in order to run app for the Least-privilege DB configuration:

- source venv/bin/activate
- export DB_PORT=5432
- export DB_NAME=sm_app
- export DB_USER=sm_app_user

- export DB_PASSWORD='ABC123'
- cd module_5/src
- python app.py

```
sm_app=# \du sm_app_user
              List of roles
  Role name  |   Attributes   | Member of
-------------+----------------+-----------
 sm_app_user | No inheritance | {}

sm_app=# SELECT grantee, privilege_type
FROM information_schema.role_table_grants
WHERE table_schema = 'public'
  AND table_name = 'applicants'
  AND grantee = 'sm_app_user';
   grantee   | privilege_type
-------------+----------------
 sm_app_user | INSERT
 sm_app_user | SELECT
(2 rows)
```

setup.py (Why Packaging Matters)

Packaging matters because it makes the project installable in a standard Python way, so imports work the same in local runs, tests, and CI. With setup.py, I can run pip install -e module_5 (or pip install -e . inside module_5) and Python will resolve modules from a consistent location instead of depending on fragile path hacks. This helps prevent “works on my machine” problems caused by different working directories or PYTHONPATH differences. Packaging also centralizes project so environment setup is more reproducible and easier to maintain. In addition, tools like uv can use setup.py metadata when syncing environments, which helps keep dependency resolution consistent. Overall, packaging makes the project easier to run, debug, share, and maintain.

Snyk Dependency Scan Findings and Remediation

I ran snyk test and found vulnerabilities. I fixed the one that had a clear fix by updating the package version (Werkzeug). After that, only one issue was left: a diskcache vulnerability that comes through llama-cpp-python. Snyk says there is currently no patch or direct upgrade for that one, so I could not fully remove it without breaking required project functionality. I documented this remaining issue as an accepted risk and included the scan screenshot (snyk-analysis.png) as proof. I also noted that the project should be scanned again later in case a fix becomes available.

Snyk Code (SAST) Summary

I ran snyk code test and it found 9 issues, all Medium (no High/Critical). Most were about file path safety and error messages showing too much internal detail. It also flagged running Flask in debug mode and one possible SSRF risk in the scraping flow. These are code-hardening issues, not package updates, so they are fixed by changing app logic and configuration rather than just updating dependencies. For this assignment, I documented the findings clearly and included the scan output as proof. Overall, there are no high-severity SAST findings, but there are medium-risk areas to improve next (input validation, safer error responses, and production-safe runtime settings).