Module 5

How to install:
All commands are run from the `module_5/` directory.

Option A: pip

```bash
python3 -m venv .venv
source .venv/bin/activate
pip install -e ".[dev]"
```

Option B: uv

```bash
uv venv
source .venv/bin/activate
uv pip install -e ".[dev]"
```

Both methods install the runtime dependencies (Flask, psycopg, beautifulsoup4)
plus dev tools (pytest, pytest-cov, pylint, pydeps).

To install only the runtime dependencies, omit the extras:

```bash
pip install -e .       # pip
uv pip install -e .    # uv
```

Dependency Graph:

App.py is the central hub of the application. Nearly every other module feeds into it, making it the top-level entry point that orchestrates all other components. Flask and its internal submodules provide the web framework layer, handling HTTP routing, request/response

lifecycle and rendering HTML templates for the user-facing interface. Scrape is responsible for fetching data from the web and it depends on bs4 (BeautifulSoup) to parse HTML and extract application information from scraped pages. Robots_checker sits upstream of scrape, indicating it enforces robots.txt compliance before any scraping takes place. Load_data handles inserting scraped records into the database and depends on both psycopg and psycopg_sql to execute parametrized SQL safely against POSTgreSQL. Cleanup_data normalizes and corrects data already in the database (such as standardizing UC university names), and also depends on psycopg, psycopg_sql and psycopg_Connection. Query_data reads and aggregates data from the database for reporting purposes, and it feeds results back to both app_py and cleanup_data, making it a shared utility across the pipeline.

SQL Injection Defenses

1. Psycopg.sql.Identifier() for every table or column name. This double-quotes and escapes the identifier according to POSTgreSQL rules. It is now structurally impossible to escape the identifier context.
2. DDL queries also use sql.Identifier(). Module_5 builds every column definition through sql.Identifier().
3. Build_insert_query() centralizes safe insert construction. Module_5 introduces build_insert_query() in load data.py which uses sql.Identifier() for the table, every column, and ON CONFLICT target, and sql.Placeholder() for each bind variable.
4. MAX_QUERY_LIMIT bounds all result sets. All SELECT queries now carry a LIMIT %s bound by MAX_QUERY_LIMIT = 1000, which caps how much data any single query can return, providing a defense-in-depth layer against data exfiltration via unbounded queries.
5. SQL construction is separated from execution. The sql.SQL(...).format(...) call only composes the query object. It never touches the database. The composed query is then passed to cur.execute(query, params) separately, where the driver binds the parameter values. This meansPython string operations are never used to inject values into SQL text, and the database driver is solely responsible for safe substitution.

Why it is safe?

Sql.Identifier() works by always wrapping the name in double quotes and escaping any internal double quotes by doubling them. This is the correct POSTgreSQL identifier quoting mechanism. There is no escape sequence that breaks out of a double-quote identifier.

Combined with %s parameterization for all values, there is no pathway for SQL injection regardless of how environment variables or scraped data are constructed.

Least Privilege DB Configuration:

Permissions granted:

- CONNECT – applicant_data database; allows app_user to open connection to the database
- USAGE – public schema; required to reference any table or sequence in the schema
- SELECT – applicants table; run_queries() runs 14 analysis queries; Kix_uc_universities() reads rows to re-normalize
- INSERT – applicants table; insert_row() adds newly scraped entries from GradCafe
- UPDATE – applicants table; Kix_gre_aw() nulliKies invalid scores; Kix_uc_universities() corrects campus names
- USAGE, SELECT – applicants_p_id_seg sequence; the p_id SERIAL PRIMARY KEY column auto-increments on each INSERT which requires advancing the sequence

Not granted:

- DELETE – the app never deletes rows
- TRUNCATE – the app never truncates tables
- DROP – the app never drops tables at runtime
- ALTER – the app never modiKies schema
- CREATE – the app never creates tables at runtime (load_data.py does, but that runs as a superuser during initial setup)

SQL snippet:

```
-- Least-privilege database user for the applicant_data application.
--
-- Run this script as a superuser (e.g., postgres) AFTER load_data.py
-- has created the database and table:
--
--   psql -U postgres -d applicant_data -f src/create_app_user.sql
--
-- Then configure the app to connect as app_user via environment variables:
--
--   export
DATABASE_URL="postgresql://app_user:change_me@localhost:5432/applicant_data"

-- 1. Create the application user
DO $$
BEGIN
    IF NOT EXISTS (SELECT 1 FROM pg_roles WHERE rolname = 'app_user') THEN
        CREATE USER app_user WITH PASSWORD 'change_me';
    END IF;
END
$$;

-- 2. Allow connecting to the database
GRANT CONNECT ON DATABASE applicant_data TO app_user;

-- 3. Allow usage of the public schema
GRANT USAGE ON SCHEMA public TO app_user;

-- 4. Grant only the permissions the app needs on the applicants table:
--    SELECT  — query_data.run_queries(), cleanup_data.fix_uc_universities()
--    INSERT  — app.insert_row()
--    UPDATE  — cleanup_data.fix_gre_aw(), cleanup_data.fix_uc_universities()
GRANT SELECT, INSERT, UPDATE ON TABLE applicants TO app_user;

-- 5. Allow the SERIAL primary key to auto-increment on INSERT
GRANT USAGE, SELECT ON SEQUENCE applicants_p_id_seq TO app_user;

-- Permissions NOT granted (least privilege):
--   DELETE   — the app never deletes rows
--   TRUNCATE — the app never truncates tables
--   DROP     — the app never drops tables
--   ALTER    — the app never alters schema
--   CREATE   — the app never creates tables at runtime
```

Why Packaging Matters.

Packaging matters because it solves the “it works on my machine” problem.

Without setup.py, every developer has to manually install the right version of

Flask, psycopg, beautifulsoup4, etc, and hope they match.

Setup.py declares the project as an installable Python package. This gives

- Importability –editable install means import app, import scrape, etc work from anywhere, not just when you cd into src/. No more sys.path hacks. Without packaging, Python can't find modules unless in the right directory or hack sys.path. With it, import app works everywhere
- Dependency management – install_requires and extras_require let pip resolve and install everything in one command instead of manually tracking requirements.txt. pip handles version conflicts automatically
- Reproducibility – anyone can clone the repo, run pip install -e " .[dev]", and get identical environment with all runtime + dev dependencies.
- Distribution – if you ever need to pip install the projects in a Docker container or CI, it just works. CI already uses pip install -e " .[dev]".
- Single command setup – pip install -e " .[dev]" installs everything. No guessing.
- CI reliability – Github Actions workflow uses pip install -e " .[dev]". Without setup.py, CI wouldn't know what to install.
- Separation of concerns – install_requires lists what the app needs to run. Extras_require["dev"] adds what developers need (pytest, pylint). Production deployments skip dev tools.

In short, it turns a collection of scripts into a proper python package with declared dependencies, making installation, testing and deployment reliable. The alternative – a bare requirements.txt with pip install -r requirements.txt. This doesn't make the code importable as a package, doesn't support extras and doesn't declare metadata. It's fine for scripts but not for multi-module projects.

Snyk code test found 5 medium-severity issues:

1. Path Traversal (scrape.py) – The –output CLI argument was passed directly to open(), allowing attacker to write files outside the intended directory via ../../ paths. Fixed by stripping to basename and writing only within cwd.
2. Debug Mode Enabled (app.py) – Flask was running with debug=True, which exposes an interactive debugger to anyone who can reach the server. Fixed by setting debug=False
3. Server Information Exposure (3 instances in app.py) – Exception details ({e}) were included in JSON error responses returned to the client, potentially leaking stack traces, database internals or network topology. Fixed by replacing with generic error messages while keeping the details in server-side logs only.