SQL Injection Defenses

1) Stopped building SQL with f-strings / concatenation / raw .format()

What changed

Queries are no longer assembled by inserting variables into raw SQL strings (no f"...", "..." + ..., or "....{}".format(...) on plain strings). Any formatting is done through **psycopg composition**: sql.SQL(...).format(...), not Python string formatting.

Why it's safer

String interpolation can turn user-controlled input into executable SQL (SQL injection). Composition keeps SQL structure and user data separate.

2) Added psycopg SQL composition for dynamic SQL structure

What changed

Imported sql (from psycopg import sql) and used it to build dynamic SQL parts. Added a helper (e.g., build_or_ilike_clause(...)) that builds OR-chains like: program ILIKE %s OR program ILIKE %s ... using sql.SQL, sql.Identifier, and sql.Placeholder, instead of manually writing repeated fragments.

Why it's safer

Dynamic SQL structure is built using objects designed to safely compose SQL. It prevents accidental injection and reduces poor quality handcrafted SQL text.

3) Refactored dynamic queries to use composed SQL objects

What changed

Large repeated OR blocks replaced with a composed SQL template (sql.SQL("""...""")) and format(institution_filters=...) or .format(university_filters=...) to insert the composed OR clause. Values remain parameters passed separately.

Why it's safer

The dynamic part is inserted as a composed SQL object (structure), not as raw text. User/data values remain bound parameters, so they can't change the meaning of SQL.

4) Enforced strict separation: statement construction instead of execution and parameters

What changed

Queries now store SQL under a statement key (e.g., "stmt") instead of assuming it's always a raw string. Execution is centralized so everything flows through: execute_query(connection, stmt, params). Helpers like get_query_stmt(...) and get_query_params(...) ensure call sites don't manually mix SQL and values.

Why it's safer

It becomes much harder for a future edit to "accidentally" do string interpolation. A single controlled execution path ensures params are always separate from SQL.

5) Converted variable components correctly (Identifiers vs Values)

What changed

Table names and column names are inserted using sql.Identifier(...). Data values use placeholders (%s or sql.Placeholder()) and are passed in params tuples.

Why it's safer

Identifiers must be treated as SQL structure; Identifier safely quotes them. Values must never be string-inserted; placeholders keep them as data only.

6) Added LIMIT to every query

What changed

Every query now includes an inherent LIMIT (either LIMIT %s or composed LIMIT {placeholder}) with MIN_QUERY_LIMIT = 1 and MAX_QUERY_LIMIT = 100. A clamp helper added and enforced centrally (often by appending the clamped limit to params in get_query_params(...)).

Why it's safer

Prevents unbounded queries that can: overload the database, return huge sensitive datasets, and enable scraping/data exfiltration. Central enforcement means even if a query is misconfigured, the hard max still applies.

Database Hardening

Here are the priviliges granted to the app_role:

```
CREATE ROLE <APP_ROLE> LOGIN PASSWORD '<APP_PASSWORD>' NOSUPERUSER NOCREATEDB NOCREATEROLE NOREPLICATION;
GRANT CONNECT ON DATABASE <DB_NAME> TO <APP_ROLE>;
GRANT USAGE ON SCHEMA public TO <APP_ROLE>;
GRANT SELECT, INSERT ON TABLE public.applicants TO <APP_ROLE>;
GRANT USAGE, SELECT ON SEQUENCE public.applicants_p_id_seq TO <APP_ROLE>;
```

I configured a dedicated least-privilege PostgreSQL application role for runtime access. The role is explicitly non-privileged (`NOSUPERUSER`, `NOCREATEDB`, `NOCREATEROLE`, `NOREPLICATION`) and is not granted ownership-level capabilities. I granted only the minimum permissions required by this app's actual behavior: `CONNECT` on the target database, `USAGE` on schema `public`, `SELECT` and `INSERT` on `public.applicants`, and `USAGE`/`SELECT` on `public.applicants_p_id_seq` to support inserts into the `SERIAL` primary key. I intentionally did not grant `DROP`, `ALTER`, `TRUNCATE`, or broad schema/table privileges. This reduces blast radius if credentials are exposed and enforces least privilege by allowing only required read/insert operations.

Python Dependency Graph (pydeps + Graphviz)

The dependency graph centers on flask_app.py, which imports the routing package (blueprints/blueprints.dashboard) and Flask entry points like flask.Flask. It also shows all environment dependencies from python-dotenv (dotenv and dotenv.load_dotenv) plus pathlib.Path, which are used to load config values and handle file paths at startup. The module blueprints.dashboard is a key functional module and is connected to Flask request/response helpers (flask.request, flask.render_template, flask.jsonify) that power web route handling and template/API responses. Database access is another major branch: psycopg and psycopg.OperationalError connect into query_data, load_data, and applicant_insert, indicating those modules execute PostgreSQL operations and handle connection/query failures. Those data-layer modules then feed into blueprints.dashboard, so the dashboard routes depend on successful query/insert/load flows. The graph also includes nearby internal Flask and dotenv submodules, which helps show not just direct imports but the immediate framework supporting the app.

Reproducible Environment and Packaging

The setup.py file defines the project as an installable Python package. It specifies metadata such as package name and version, where source code is located (src layout), required runtime dependencies, and supported Python versions. Packaging ensures imports behave consistently across local development, testing, and CI environments. Using editable installs (pip install -e .) reduces path-related issues and prevents “it works on my machine” errors. The setup.py module defines how the project itself is installed and distributed.

Fresh Install:

- Method 1 (pip):

1) python -m venv .venv

2) .\.venv\Scripts\Activate.ps1

3) python -m pip install --upgrade pip

4) python -m pip install -r requirements.txt

- Method 2 (uv):

1) uv venv .venv

2) .\.venv\Scripts\Activate.ps1

3) uv pip sync requirements.txt