SQL Injection Defenses

What changed:

All SQL queries across module_5 were migrated from raw string formatting to psycopg's sql composition module. Specifically:

1. No raw string interpolation in SQL. Every query that previously used f-strings, string concatenation, or .format() was rewritten. SQL text is now wrapped in sql.SQL(...) objects from the psycopg.sql module.
2. Centralized query builder with enforced LIMIT. A utility function build_query() in sql_utils.py wraps every SELECT query in sql.SQL and appends a LIMIT clause. The limit value is clamped to the range 1-100 via clamp_limit(), preventing any caller from requesting unbounded result sets. The limit is injected as sql.Literal(clamped_value), not as a raw integer in a string.
3. Parameter binding for all user-sourced values. INSERT statements use %s placeholders with a separate params tuple passed to cursor.execute(). This means data values (from scraped JSON, for example) are never embedded in the SQL text -- the database driver handles escaping and quoting.
4. Dynamic identifiers use sql.Identifier. Where table or column names could vary, sql.Identifier(...) is used so that psycopg properly quotes them, preventing identifier-based injection.

Why it is safe:

- The sql.SQL / sql.Literal / sql.Identifier API guarantees that every component of a query is type-checked and properly escaped by psycopg before being sent to PostgreSQL. An attacker cannot break out of a quoted literal or identifier because the driver controls the quoting, not string concatenation.
- Parameter binding (%s + params) uses the database wire protocol's native parameterized-query support, which separates the SQL command from the data at the protocol level -- SQL injection is structurally impossible through this path.
- The enforced LIMIT cap (max 100 rows) prevents data exfiltration even if a future code change accidentally exposes a query to user input: the worst case is 100 rows, not the entire table.
- The Flask endpoints (/, /update_analysis, /pull_data) currently accept no user-supplied parameters that flow into SQL, providing defense in depth.

Least-Privilege Database Configuration

Recommended permissions:

The application interacts with PostgreSQL through two distinct roles, each granted only the minimum privileges required:

1. Application role (used by app.py, query_data.py at runtime):
   - CONNECT on the target database
   - USAGE on the public schema
   - SELECT on the applicants table

- No INSERT, UPDATE, DELETE, or DDL privileges

This role can only read data. Even if the Flask app were compromised, an attacker could not modify or destroy data through this connection.

2. Pipeline role (used by load_data.py, load_new_data.py, scrape.py):
   - CONNECT on the target database
   - USAGE, CREATE on the public schema (for table creation in initial load)
   - SELECT, INSERT on the applicants table
   - DROP TABLE only for the initial bulk-load script (load_data.py)

   This role can insert new records and, during initial setup, recreate the table. It should NOT have SUPERUSER, CREATEDB, or CREATEROLE privileges.

Why this matters:

- Blast radius reduction. If the web-facing Flask app is compromised, the attacker inherits only SELECT privileges -- they cannot DROP tables, INSERT malicious rows, or escalate to other databases.
- Separation of duties. The pipeline scripts that modify data run on a schedule or on-demand behind pipeline_lock, not in response to arbitrary web requests. Their elevated INSERT permission is isolated from the public-facing read path.
- No superuser access. Neither role should use the PostgreSQL postgres superuser account in production. The .env.example template prompts operators to configure dedicated credentials (DB_USER=your_username) rather than reusing the default superuser.
- Environment-based secrets. Database credentials are loaded from .env (gitignored) via python-dotenv, never hardcoded in source. The .env.example file documents required variables without exposing real values.

Example SQL to create these roles:

```
-- Read-only role for the Flask dashboard
CREATE ROLE app_reader LOGIN PASSWORD 'strong_password_here';
GRANT CONNECT ON DATABASE mydb TO app_reader;
GRANT USAGE ON SCHEMA public TO app_reader;
GRANT SELECT ON applicants TO app_reader;

-- Read-write role for the data pipeline
CREATE ROLE pipeline_writer LOGIN PASSWORD 'another_strong_password';
GRANT CONNECT ON DATABASE mydb TO pipeline_writer;
GRANT USAGE, CREATE ON SCHEMA public TO pipeline_writer;
GRANT SELECT, INSERT ON applicants TO pipeline_writer;
```

Dependency Graph Summary

It's a dependency graph generated by pydeps showing every Python module that app.py imports, and their internal sub-module dependencies. It was rendered by Graphviz.
Each arrow means "is imported by" (dependency flows toward app.py at the bottom).
For example: dotenv → app_py = app.py imports dotenv
Ellipses = leaf/simple modules; Rectangles (blue) = framework packages (flask, psycopg)
app.py sits at the bottom as the root — everything flows down to it

In summary, The SVG visualizes: app.py depends on 3 libraries — dotenv for config, flask for the web server, and psycopg for PostgreSQL access — and shows the full internal dependency tree of each library.

Explanation of setup.py

Purpose:

The setup.py file is the standard configuration script for packaging, distributing, and installing the module_5 Python project. It ensures that the software can be easily installed into a Python environment along with its required dependencies.

Why packaging matters:

Packaging matters because it transforms a loose collection of Python scripts into a reproducible, installable unit. With a setup.py, anyone can run pip install and get the exact same dependency versions, entry points, and module structure — regardless of their machine. Without it, you're relying on manual instructions ("copy these files, install these libraries, run from this directory"), which leads to "works on my machine" problems. Packaging also enables proper dependency resolution (pip handles version conflicts), clean imports (no sys.path hacks), and distribution — whether to PyPI, a private registry, or for just another team mate.

How it Works:

The script imports setup and find_packages from the setuptools library and calls the setup() function with the following key configurations:

1. Metadata: Defines the project's identity, including:
    - name: "module_5"
    - version: "1.0.0"
    - description and author information.

2. Package Discovery: The packages=find_packages() argument automatically scans the directory structure for Python packages (directories containing __init__.py files) to include in the distribution.

3. Dependency Management: The install_requires list defines external libraries that pip will automatically install when this project is installed:
    - psycopg[binary]: The PostgreSQL database adapter.
    - python-dotenv: A utility to load environment variables from .env files.
    - Flask: The web framework used for the dashboard (constrained to versions >=2.3 and <4).

4. Environment Constraints: The python_requires=">=3.10" argument enforces that the project is only installed on Python 3.10 or newer, preventing compatibility issues with older interpreters.

How to Install and Run

Using pip:

```
# From the repository root, install module_5 in editable (development) mode:
pip install -e module_5/

# Or install with all dev/test dependencies:
pip install -r module_5/requirements.txt
pip install -e module_5/
```

```
# Run the Flask app:
python -m module_5.src.app

# Run the pipeline:
python -m module_5.src.run_pipeline

# Run tests:
pytest module_5 -v
```

Using uv:

```
# Create a virtual environment and install in one step:
uv venv
uv pip install -e module_5/

# Or install all dependencies from requirements.txt:
uv pip install -r module_5/requirements.txt
uv pip install -e module_5/

# Run the Flask app (through the managed environment):
uv run python -m module_5.src.app

# Run tests:
uv run pytest module_5 -v
```

Note: uv is a drop-in replacement for pip that resolves and installs dependencies significantly faster. The commands mirror pip but are prefixed with uv pip for install operations and uv run for execution.