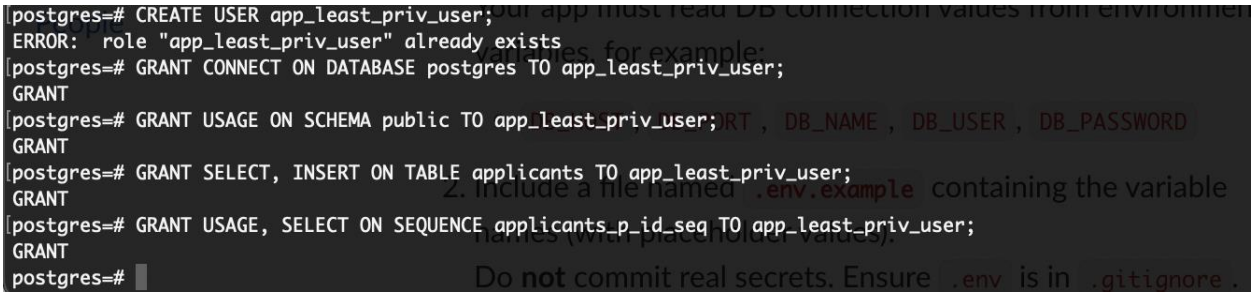
Step 2:

In this module, every database query was refactored to replace unsafe Python f-strings and string concatenation with psycopg SQL composition. By utilizing sql.SQL, sql.Identifier, and sql.Placeholder, the application now strictly separates the SQL statement construction from the data execution phase. Dynamic components like table and column names are explicitly quoted via sql.Identifier, while user-supplied values are passed as bound parameters in the cursor.execute() call.

This approach is safer because it prevents malicious input from being interpreted as executable code by the PostgreSQL engine. A mandatory inherent LIMIT was also integrated into every query, enforced by logic that clamps user-requested results to a range of 1–100. This multi-layered defense ensures the application is resilient against both classic SQL injection attacks and potential data exfiltration through mass-result requests.

Step 3:

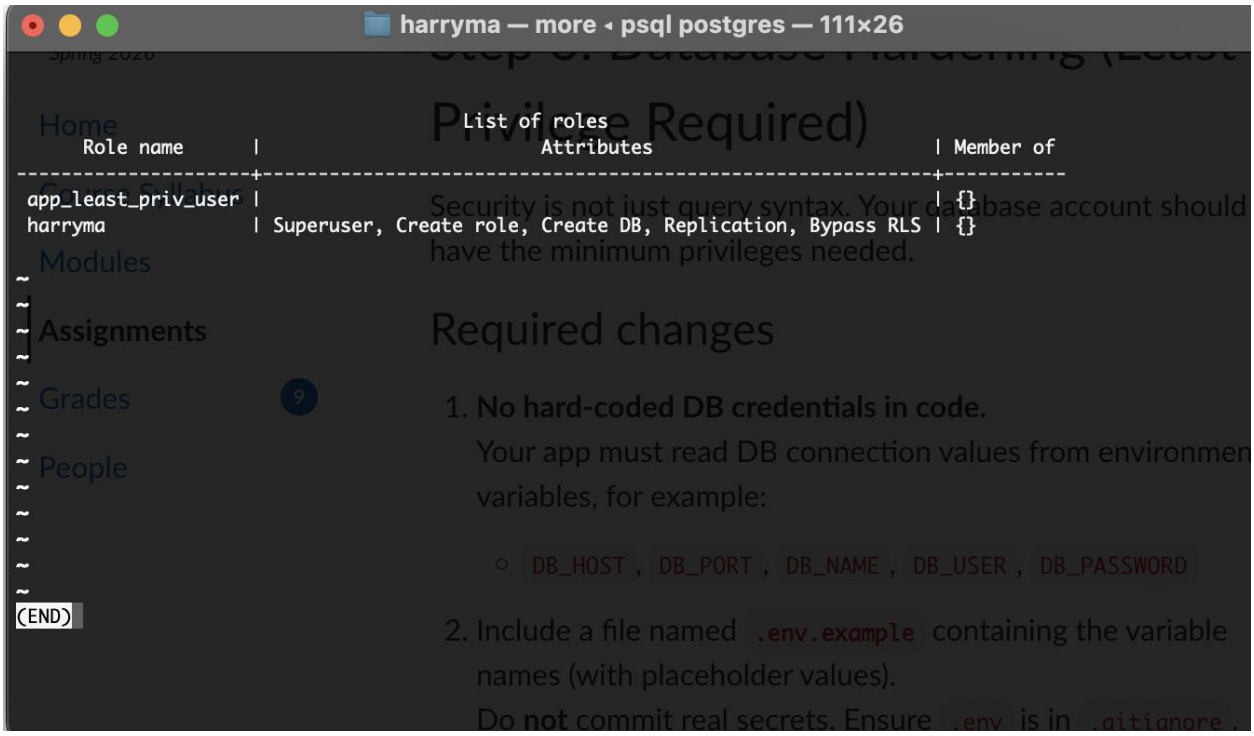


GRANT CONNECT… allows the user to connect to the database

GRANT USAGE ON SCHEMA… allows the user to access objects stored in the public schema (where ‘applicants’ table is located)

GRANT SELECT/INSERT…. SELECT allows the reader to read existing data points in the table and INSERT allows the user to add new datapoints to the table(Flask needs to pull data to be displayed and add new data points when scrape.py is iterated)

GRANT USAGE/SELECT… SEQUENCE… grants permission to interact with the autogenerated p_id primary key (needed to prevent permission errors when adding new data if p_id is untouchable)



Step 4:

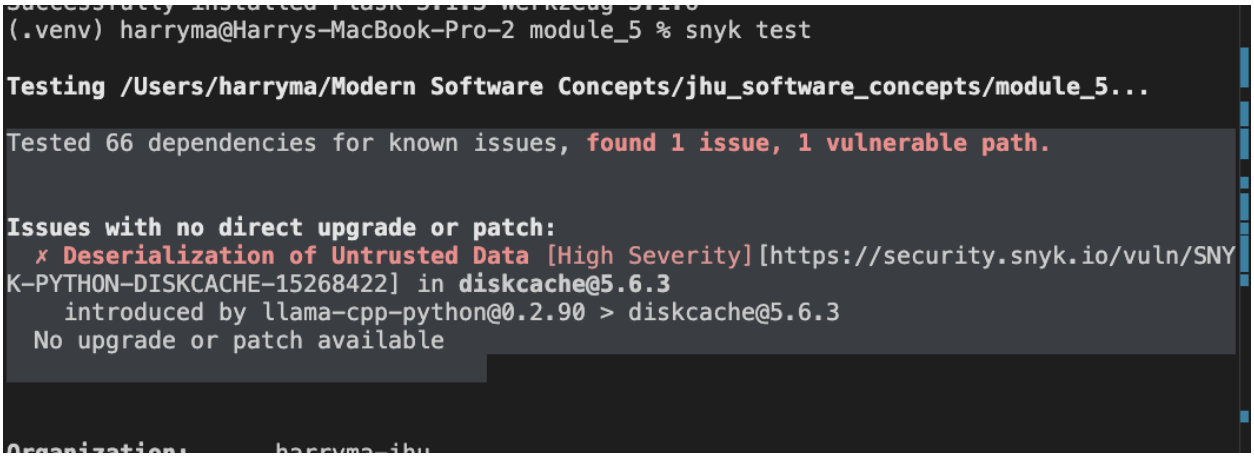
Three main dependencies feed into the core application: python-dotenv, Flask, and psycopg. Dotenv acts as the foundational layer for security, loading sensitive credentials from environment variables to ensure the DATABASE_URL is never hard-coded, which is a key requirement for portable CI/CD environments. Flask serves as the primary web framework, utilizing Blueprints for modular routing and Jinja2 templating to render the analysis page with the required "Answer:" labels and two-decimal formatting. Within the Flask ecosystem, the jsonify utility is critical for implementing "busy-state" behavior, allowing the API to return specific 409 Conflict status codes when a data pull is already in progress. To manage the data layer, psycopg2 provides the necessary adapter for PostgreSQL communication, using connection objects and cursors to execute the idempotent INSERT queries that prevent duplicate records. Together, these dependencies enable a testable architecture where the database logic, web routing, and environment configuration remain decoupled and secure.

Step 5:

Adding a setup.py file to the project directory is a critical step for ensuring long-term maintainability and professional distribution. This configuration file effectively transforms a collection of individual scripts into a formal Python package, allowing for a standardized installation process across various environments. By defining the project structure this way, developers can ensure that internal module imports remain consistent whether the code is executing on a local workstation, within a testing suite, or throughout a continuous integration pipeline. Furthermore, the inclusion of a setup.py facilitates editable installs, which allows the environment to reflect code changes in real time while maintaining correct system paths. This practice also enables modern dependency managers like uv to accurately synchronize the environment, providing a robust foundation for reproducible research and development.

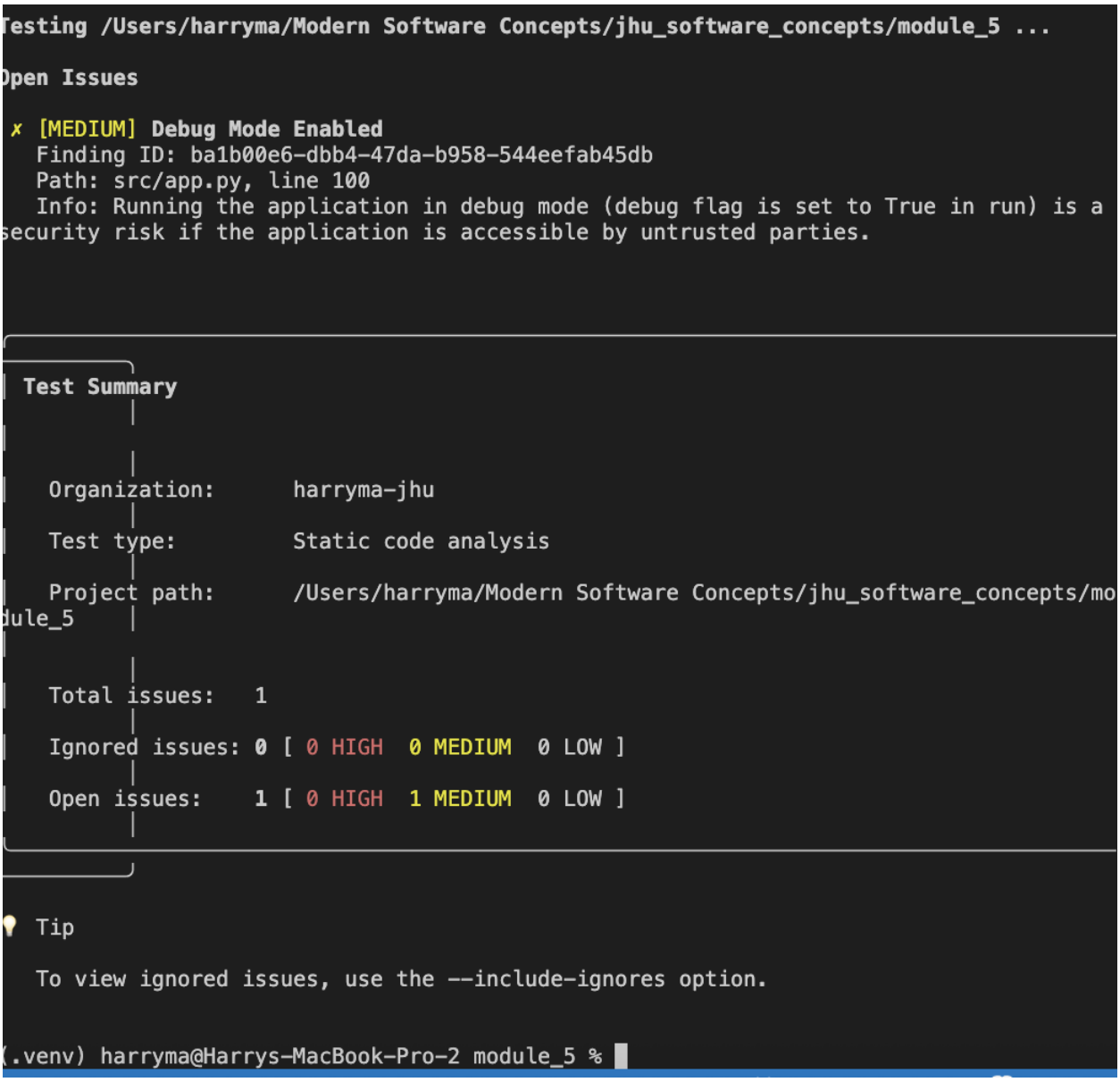
Step 6:

Initial run:



Fixed: (.snyk file created)

Extra credit (Snyk Code):



This makes sense since debug is still set to True in app.py to help with instant feedback on the webpage with changes I've made in the code. Eventually, this will be turned to False and the 1 open issue will be resolved.