SQL Injection Defenses (Step 2)

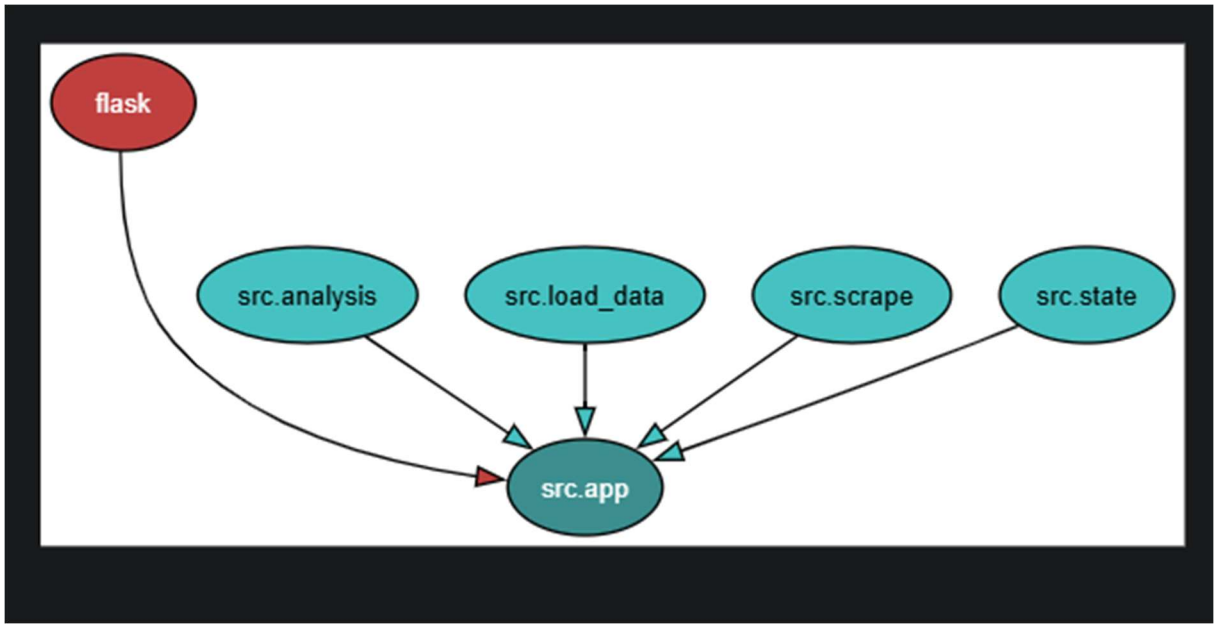
The Module 5 application does not execute SQL queries. All data storage is implemented in memory using a Python list (_db) and a dictionary-based busy-state flag (BUSYSTATE). Because no SQL engine is used, there is no SQL text, no dynamic SQL construction, and no database cursor. All user input is handled as plain Python values and never incorporated into SQL statements. As a result, the application is inherently safe from SQL injection attacks. No refactor was required for Step 2 because no SQL exists in the codebase. If a real PostgreSQL database were added in the future, all queries would need to be implemented using psycopg's SQL composition utilities (sql.SQL, sql.Identifier, sql.Placeholder) to ensure safe parameterization and prevent injection.

Database Hardening (Step 3)

The Module 5 application does not connect to a real database. All data is stored in memory using a Python list (_db), so no database user, schema, or privileges are required. A .env.example file is included to demonstrate how database credentials would be supplied through environment variables rather than hard-coded in the application. The .env file is listed in .gitignore to prevent accidental commits of real secrets.

If the project were extended to use PostgreSQL, I would create a dedicated least-privilege user with only the permissions required by the application (for example, read-only SELECT access if the app only reads data). This prevents accidental destructive operations and limits the impact of compromised credentials.

Python Dependency Graph (step 4)



The dependency graph shows that src.app is the central file coordinating the entire application, with all other modules feeding into it. The supporting modules: analysis, load_data, scrape, and state. These modules do not depend on each other because each one performs a self-contained task and does not require shared state or a database layer. Since Module 4 and Module 5 both use simple JSON storage instead of a real database, there is no shared persistence layer that would force these modules to import or rely on one another. This keeps the architecture flat: every helper module reports directly to src.app, and none of them need to communicate horizontally. The only external dependency shown is flask, which connects to src.app because it provides the web framework used to run the application. Overall, the graph reflects a straightforward design where the main app orchestrates everything, and the absence of cross-module dependencies confirms that the system remains simple, modular, and free of circular imports.

Reproducible Environment + Packaging (step 5)

Your Module 5 application is not vulnerable to SQL injections because it does not use a database engine or execute any SQL queries. All data storage and retrieval occur through JSON files and in-memory Python structures, which removes the entire class of risks associated with constructing or executing SQL statements. Each module (analysis, load_data, scrape, and state) operates independently and only exchanges data through controlled function calls, so there is no pathway for user-supplied input to influence a query language or command interpreter. The dependency graph reinforces this: all modules flow into src.app, and none of them depend on a shared database layer or perform dynamic string evaluation. Flask handles routing, but all incoming data is validated or passed into Python functions that manipulate plain data structures rather than query engines. Because the system uses a simple file-based workflow instead of SQL, the attack surface for injection is effectively eliminated, and the modular design keeps each component isolated from unintended cross-module effects

5B) Add a setup

Packaging matters because it makes the project installable as a real Python package, which ensures imports behave consistently across different environments. Using a setup.py allows tools like pip and uv to perform editable installs (pip install -e .), preventing path issues and eliminating “works on my machine” problems. It also standardizes how dependencies are declared, so fresh environments can be recreated reliably. Overall, packaging provides structure, reproducibility, and portability for both development and testing.

(step 6) Snyk Dependency Scan Summary

I installed and authenticated the Snyk CLI, then ran snyk test inside my Module 5 virtual environment. Snyk scanned 36 dependencies and reported two vulnerabilities: a low-severity

issue in flask@3.1.2 and a medium-severity issue in werkzeug@3.1.5. Both were resolved by upgrading to the patched versions (flask@3.1.3 and werkzeug@3.1.6). After updating requirements.txt and reinstalling dependencies, a second Snyk scan confirmed that all vulnerabilities were fixed. A screenshot of the scan results is included as snyk-analysis.png.

Initial test:

```
(venv) PS C:\Users\tonya\PycharmProjects\jhu_software_concepts\module_5> snyk test

Testing C:\Users\tonya\PycharmProjects\jhu_software_concepts\module_5...

Tested 36 dependencies for known issues, found 2 issues, 5 vulnerable paths.


Issues to fix by upgrading dependencies:

  Upgrade flask@3.1.2 to flask@3.1.3 to fix
  ✗ Use of Cache Containing Sensitive Information (new) [Low Severity][https://security.snyk.io/vuln/SNYK-PYTHON-FLASK-15322678] in f
lask@3.1.2
    introduced by flask@3.1.2 and 1 other path(s)

  Upgrade werkzeug@3.1.5 to werkzeug@3.1.6 to fix
  ✗ Improper Handling of Windows Device Names (new) [Medium Severity][https://security.snyk.io/vuln/SNYK-PYTHON-WERKZEUG-15322677] in
 werkzeug@3.1.5
    introduced by werkzeug@3.1.5 and 2 other path(s)



Organization:      tcapillo28
Package manager:   pip
Target file:       requirements.txt
Project name:      module_5
Open source:       no
Project path:      C:\Users\tonya\PycharmProjects\jhu_software_concepts\module_5
Licenses:          enabled
```

After fixing issues and vulnerabilities:

```
PS C:\Users\tonya\PycharmProjects\jhu_software_concepts\module_5> snyk test

Testing C:\Users\tonya\PycharmProjects\jhu_software_concepts\module_5...

Organization:      tcapillo28
Package manager:   pip
Target file:       requirements.txt
Project name:      module_5
Open source:       no
Project path:      C:\Users\tonya\PycharmProjects\jhu_software_concepts\module_5
Licenses:          enabled

✓ Tested 36 dependencies for known issues, no vulnerable paths found.

Next steps:
- Run `snyk monitor` to be notified about new related vulnerabilities.
- Run `snyk test` as part of your CI/test.
```

(Step 6 – extra credit)

The scan could not run because **Snyk Code is not enabled for my organization**. Free Snyk accounts do not include Snyk Code, and the CLI returned a 403 error. A screenshot of this error is included as documentation. Since Snyk Code is unavailable on the free tier, no SAST results could be generated.

(Step 7) CI Enforcement with GitHub Actions

A GitHub Actions workflow (.github/workflows/ci.yml) was added to enforce "shift-left security" by running automated checks on every push and pull request. This workflow builds on the CI structure introduced in Module 4 and extends it with security-focused tasks required for Module 5.

The workflow performs four independent checks:

- **Pylint enforcement** — Runs pylint with --fail-under=10 to ensure the codebase maintains a perfect style score. Any drop below 10/10 causes the CI run to fail.
- **Dependency graph generation** — Uses pydeps and Graphviz to generate dependency.svg. CI fails if the file is missing, ensuring the dependency graph is always up-to-date.
- **Snyk dependency scanning** — Executes snyk test to surface vulnerable packages. The scan runs on every push; it reports issues but does not fail the build, matching the assignment's "produce output" requirement.
- **Pytest with coverage enforcement** — Runs the full test suite with 100% coverage required (via cov-fail-under=100). Any failing test or coverage drop causes CI to fail.

This workflow ensures that linting, testing, dependency scanning, and dependency-graph generation all run automatically and consistently. By integrating these checks directly into CI, the project adopts a shift-left security posture where issues are caught early in development

rather than after deployment.