Module 4: Pytest and Sphinx

JHU EP 605.256 – Modern Software Concepts in Python

Introduction

In this assignment you will write an automated test suite for the Grad‑Café system you built in Module 3—covering your Flask page, interactive buttons, analysis outputs/formatting, and database writes. You will follow a lightweight TDD loop and publish documentation with Sphinx.

Skills:Pytest, Flask testing, HTML parsing, database testing, CI hygiene, Documentation

Assignment Overview

In this assignment you will:

1. Write a Pytest suite for your Grad Café analytics application that verifies:
    a. The Flask Analysis page loads and renders required components
    b. The "Pull Data" and "Update Analysis" buttons behave correctly (including doing nothing when a pull is in progress)
    c. Analysis outputs are labeled and percentages are shown with two decimals
    d. New rows are correctly written into PostgreSQL using the Required schema
2. Ensure 100% test coverage and automatically runs tests using Github Actions.
3. Documents the service with Sphinx: include developer setup, architecture, API reference for your modules, and a "How to run tests" page; publish the HTML to Read the Docs

After this assignment, you will have an extensible, test-driven, documented Grad Café data and web analysis service that other developers can run, test, and extend confidently.

1. Write a Pytest Suite for your Grad Cafe Analytics Application

Over the past two weeks, we've developed out our Grad Cafe Analytics system. So far our system has two core subsystems:

1. Web (Flask): serves an Analysis page with two buttons: "Pull Data" and "update Analysis"
2. Data (ETL + DB): scraping/cleaning/loading rows into PostgreSQL and computing summary analysis for display.

Before we continue to develop our system, we want to ensure stability of our system – to make sure that the fundamental capabilities do not break as we continue to grow functionality. So, let's add in some tests to ensure the functionality works as expected now and can be tested in the future.

First, create a copy of your module_3 folder and name the new folder module_4. Move all existing code into a new folder called "/src" and then add a new folder (at the same level as source) called "/tests".

Within the "tests" folder, You need to write the following unit tests:

1. **Flask App & Page Rendering –** "tests/test_flask_page.py"
    a. Test app factory / Config: Assert a testable Flask app is created with required routes (e.g. should test each of your "/routes" that you establish in flask).
    b. Test GET /analysis (page load)
        i. Status 200. ii. Page Contains both "Pull Data" and "Update Analysis" buttons
        iii. Page text includes "Analysis" and at least one "Answer:"
2. **Buttons & Busy-State Behavior –** "tests/test_buttons.py"
    a. Test POST /pull-data (or whatever you named the path posting the pull data request) i. Returns 200 ii. Triggers the loader with the rows from the scraper (should be faked / mocked)
    b. Test POST /update-analysis (or whatever you named the path posting the update analysis request)
        i. Returns 200 when not busy
    c. Test busy gating
        i. When a pull is "in progress", POST /update-analysis returns 409 (and performs no update).
        ii. When busy, POST /pull-data returns 409
3. **Analysis Formatting –** "tests/test_analysis_format.py"
    a. Test labels & Rounding
        i. Test that your page include "Answer" labels for rendered analysis
        ii. Test that any percentage is formatted with two decimals.
4. **Database writes –** "tests/test_db_insert.py"
    a. Test insert on pull
        i. Before: target table empty
        ii. After POST/pull-data new rows exist with required (non-null) fields
    b. Test idempotency / constraints
        i. Duplicate rows do not create duplicates in database (accidentally pulling the same data should not result in the database acquiring duplicated rows).
    c. Test simple query function
        i. You should be able to query your data to return a dict with our expected keys (the required data fields within M3). **5. Integration Tests** "tests/test_integration_end_to_end.py"
    a. End-to-end (pull -> update -> Render)
        i. Inject a fake scraper that returns multiple records
        ii. POST /pull-data succeeds and rows are in DB
        iii. POST /update-analysis succeeds (when not busy)
        iv. GET /analysis shows updated analysis with correctly formatted values b. Multiple pulls
        i. Running POST /pull-data twice with overlapping data remains consistent with uniqueness policy.

Marks – must be listed in a pytest.ini

All test must be marked with one (or more) of the following Pytest markers:

- @pytest.mark.web — page load / HTML structure.
- @pytest.mark.buttons — button endpoints & busy-state behavior.
- @pytest.mark.analysis — labels and percentage formatting.
- @pytest.mark.db — database schema/inserts/selects.
- @pytest.mark.integration — end-to-end flows.

Policy: There should not exist a test that is unmarked. Running

pytest -m "web or buttons or analysis or db or integration"
must execute the entire suite.

You must use pytest-cov (plugin discussed in class) to guarantee 100% coverage of all code within your repository across all modules. This may require you to write additional tests specific to your code design. Please extend test suite as necessary for your implementation.

Your pytes.ini file should look like the following:

‘’’

[pytest] addopts = -q --cov=module_4/src --cov-report=term-missing --cov-fail-

under=100 markers =   web: Flask route/page tests   buttons: "Pull Data" and

"Update Analysis" behavior   analysis: formatting/rounding of analysis output

db: database schema/inserts/selects   integration: end-to-end flows

‘’’

Please commit the terminal coverage summary to “module_4/coverage_summary.txt”

In lecture we will look at github actions, and how to create a simple CI pipeline. For this assignment, you should provide (and screenshot evidence of it running and succeeding) a minimal GitHub Actions workflow that starts Postgres and runs your pytest suite. Please name your file showing successful github actions “actions_success.png”. Place your workflow file under “.github/workflows/tests.yml”

2. Sphinx Documentation

Within the lecture, we provided instruction on how to integrate a github project with sphinx documentation. Follow this process to connect your github repo, and then build and publish sphinx documentation for your grad cafe application.

Produce Sphinx docs that cover:

- **Overview & setup:** how to run the app, required environment vars (e.g., DATABASE_URL), and how to run tests.
  Architecture: brief description of the web, ETL, and DB layers and their responsibilities.
- **API reference:** autodoc pages for your key modules (scrape.py, clean.py, load_data.py, query_data.py, flask_app.py routes).
- **Testing guide:** how to run marked tests, expected selectors, and any test doubles/fixtures you provide.

Your module_4 folder should look like the following:

```
module_4/
  src/               # your application code (Flask, ETL/DB, queries)
  tests/             # ALL test code lives here
  pytest.ini
  requirements.txt
  README.md
  docs/              # Sphinx project (source + conf.py)
```

(With proof of work pngs added)

3. Assignment Requirements: SHALL/SHOULD/SHALL NOT

SHALL:

Pytest suite coverage

- Verify GET /analysis returns 200 and renders required components: page title/text (“Analysis”), “Pull Data” and “Update Analysis” buttons, and at least one “Answer:” label.
- Verify POST /pull-data returns 200/202 with {"ok": true} when not busy and triggers the loader (use fake/mocked scraper/loader in tests).
  Verify busy gating: when a pull is in progress, POST /update-analysis returns 409 with {"busy": true} and performs no update; same for POST /pull-data if you choose to gate both.
- Verify analysis formatting: any percentage rendered on the page is shown with two decimals (e.g., 39.28%) and analysis items are labeled with “Answer:”.
- Verify database writes: after POST /pull-data, new rows are inserted into PostgreSQL with the required Module-3 schema and required (non-null) fields.
- Verify idempotency/constraints: duplicate pulls do not duplicate rows (respect your uniqueness policy).
- Verify query function returns the dictionary of expected keys used by the analysis template (Module-3 required fields).

- Provide at least one end-to-end test: pull ⟶ update ⟶ render shows updated analysis with correct formatting.

Test organization & markers

- All tests shall be marked with one (or more) of: web, buttons, analysis, db, integration.
- Running pytest -m "web or buttons or analysis or db or integration" shall execute the entire suite; no unmarked tests are permitted.

App testability

- Expose a Flask create_app(...) factory.
- Provide stable selectors for UI tests (e.g., data-testid="pull-data-btn" and data-testid="update-analysis-btn").
- Use DATABASE_URL to connect to PostgreSQL (tests may override).

Documentation (Sphinx)

- Publish Sphinx docs that include: Overview & setup (env vars, how to run app/tests), Architecture (Web/ETL/DB roles), API reference (autodoc for scrape.py, clean.py, load_data.py, query_data.py, Flask routes), and a Testing guide (markers, selectors, fixtures).
  Publish HTML (e.g., Read the Docs) and link it in the repo README.

SHOULD

- **Testing ergonomics**

  - Use dependency injection so tests can pass fake scraper/loader/query functions without hitting the network.
    Use BeautifulSoup (or similar) for HTML assertions and a regex for two-decimal percentages.
  - Add negative/error-path tests (e.g., loader error yields non-200 and no partial writes).

- **Quality & CI**

  - Provide a minimal GitHub Actions workflow that starts Postgres and runs pytest. Keep tests fast and deterministic (seconds, not minutes).

- **Docs polish**

  - Add an “Operational notes” page (busy-state policy, idempotency strategy, uniqueness keys).
    Include a troubleshooting section for common local/CI issues.

SHALL NOT

- **Testing pitfalls**
    - Shall not ship any test that depends on live internet or long-running scrapes.
    - Shall not use arbitrary `sleep()` for busy-state checks (make state observable/injectable).
    - Shall not leave any test unmarked (policy requires marks).

- **Behavior / formatting**
    - Shall not render percentages with varying precision; analysis percentages must be two decimals.
    - Shall not change the Module-3 schema or required fields in a way that breaks prior functionality or tests.

- **Security / portability**
    - Shall not hard-code secrets, credentials, or environment-specific paths in code or tests. Shall not require manual UI interaction for tests (no browser clicking; use Flask test client).

4. Deliverables

1. The SSH URL to your GitHub repository
2. README under **`module_4`**
3. requirements.txt under **`module_4`**
4. Sphinx generated HTML associated with your application under **`module_4`**
5. Proof of coverage under coverage_summary.txt under **`module_4`**
6. Proof of successful GitHub s CI with actions_success.png Action (for an image of a green run) and a ".github/workflows/tests.yml" under **`module_4`**
7. Link to sphinx read the docs documentation
8. All listed test files (see above) under **`module_4`**
9. Please remember to submit to both canvas and commit to your private github! **Please let us know if you have any questions via Teams or email!**