AIDE505 · MLOPS · LAB 1

Unit and Integration Testing of a Flask Application

Automated tests with pytest, fixtures and the Flask test client

Hadi Hijazi
Lebanese University, Master 2 AIDE
November 2026

Abstract. This lab builds an automated test suite for a small Flask registration and login application. Six provided tests (three unit, three integration) were run and studied, and four new tests were added to answer the exercises. The final suite has 13 test cases, which all pass in about 0.2 s and cover 100% of the business-logic module and 89% of the web layer. A fault-injection experiment showed that only the unit tests detect a weakened password rule, which supports testing business rules directly instead of only through HTTP.

1. Objective

The goal of the lab is to learn the difference between unit and integration tests by writing both for the same application. Unit tests call one function or class directly. Integration tests send HTTP requests through Flask, the templates, the session and the business logic together. The lab also introduces pytest fixtures, parametrised tests, fault injection and coverage measurement.

2. System under test

The application has two modules. `auth.py` holds the business logic and has no dependency on Flask: input validation (`validate_username`, `validate_password`), password hashing with PBKDF2-HMAC-SHA256 and a random 16-byte salt (`hash_password`, `verify_password`), and an in-memory `UserStore` that normalises usernames to lower case. `app.py` is the web layer. It exposes `/register`, `/login`, `/welcome` and `/logout`, stores the logged-in user in a signed session cookie, and reports errors with flash messages and HTTP status codes (400 for invalid registration, 401 for failed login).

`app.py` builds the application through a factory, `create_app(store=None, testing=False)`. Because the store can be passed in, a test can create a fresh store, give it to the application, and later inspect the same object to check what the web layer did to the backend.

3. Environment

All tests were run on an Apple M1 Pro laptop running macOS 27, with Python 3.12.14 in a virtual environment created with `uv`, Flask 3.1.3, pytest 9.2.0 and pytest-cov 7.1.0. The system interpreter (Python 3.9) could not import the project: `auth.py` uses the type annotation `bytes | None`, which requires Python 3.10 or later, and the import failed with a `TypeError`. The guide states Python 3.10+ as a requirement, so a Python 3.12 environment was used for all runs.

When registering with the password `abc` in the running application, the page shows "Password must be at least 8 characters". The message is raised as an `AuthError` by `validate_password()` in `auth.py`,

called from `UserStore.register()` after the username has been validated. The route catches the exception, flashes its text and returns status 400.

4. Test design

`tests/conftest.py` defines two fixtures. `store` returns a new, empty `UserStore` for every test, so tests cannot affect each other. `client` depends on `store`, builds the application with that same store and yields Flask's test client, which sends requests to the application without starting a server and keeps cookies between requests like a browser.

Table 1. Test cases in the final suite. Rows marked "added" were written for the exercises.

Test	Level	What it verifies
`test_hash_and_verify_password`	unit	the hash never contains the plain password; correct and wrong passwords verify correctly
`test_hash_password_is_salted` (added)	unit	the same password hashes to a different string each time, and both still verify
`test_register_rejects_invalid_input` ×5	unit	too-short, too-long and illegal usernames, short passwords and passwords without a digit raise `AuthError` and store nothing
`test_register_rejects_duplicate_username`	unit	usernames are case-insensitive, and a duplicate is rejected
`test_register_then_login_shows_welcome`	integration	register, then log in, then reach the welcome page
`test_login_with_wrong_password_fails`	integration	status 401, error message shown, no user in the session
`test_welcome_requires_login_and_logout_clears_session`	integration	`/welcome` redirects when logged out, and access ends after logout
`test_get_register_shows_form` (added)	integration	`GET /register` returns 200 with the registration form
`test_register_existing_username_fails` (added)	integration	a duplicate registration over HTTP returns 400, shows the error and stores one user

The parametrised test runs once per input row, so the provided suite of 6 functions produces 9 test cases. With the added row and the three added functions, the final suite produces 13.

5. Results

The provided suite gave `9 passed`, as expected by the guide. The final suite gives `13 passed in 0.21 s`. The integration tests are fast because the test client calls the application in-process; the slowest part of the suite is the PBKDF2 hashing (100,000 iterations), not the HTTP layer.

6. Exercises

6.1 Fault injection: weakening the password rule

`MIN_PASSWORD_LENGTH` was changed from 8 to 4 and the suite was run again. Exactly one test case failed:

```
FAILED tests/test_unit.py::test_register_rejects_invalid_input[alice-short1-at least 8]
Failed: DID NOT RAISE AuthError
```

The password `short1` has six characters and contains a digit, so a limit of 4 accepts it, and `pytest.raises` reports that the expected exception was never raised. None of the integration tests failed, because they all use valid passwords: they check that the parts are connected, not that each rule is correct. The rule is guarded only by the unit test that sits just below the boundary. The constant was restored to 8 afterwards.

6.2 Salting

The added test `test_hash_password_is_salted` hashes `secret123` twice, asserts that the two results differ, and asserts that both verify (listing in Appendix A). The result differs because `hash_password` draws a new random salt with `os.urandom(16)` on every call and stores it next to the digest.

This matters for security. Without a salt, two users with the same password would have identical hashes, which reveals shared passwords to anyone who reads the store, and precomputed tables (rainbow tables) of common passwords could reverse the hashes directly. With a per-user salt, an attacker who steals the store must attack each hash separately, and the 100,000 PBKDF2 iterations make each guess expensive.

6.3 Upper bound of the username length

The row `("a" * 21, "secret123", "Username")` was added to the parametrised test. The username rule is the regular expression `^[a-z0-9_]{3,20}$`. The existing row `"ab"` tested the lower bound, and the new row tests the value just above the upper bound, so both edges of the valid range are now covered.

6.4 Rendering the registration page

`test_get_register_shows_form` sends `GET /register` and asserts status 200 and the text `Create an account` in the response body. This covers the GET branch of the route and the rendering of `form.html` in registration mode, which the provided tests never requested directly.

6.5 Duplicate registration over HTTP

`test_register_existing_username_fails` registers `bob`, registers `bob` again with another password, and asserts status 400, the message `already taken` in the page, and `len(store) == 1`. The unit test in Table 1 already checked the rule itself; this test checks the wiring around it: that the route catches the `AuthError`, uses the right status code and shows the message to the user.

6.6 Coverage

Table 2. Line coverage of the final suite (`pytest --cov=. --cov-report=term-missing`).

File	Statements	Missed	Coverage	Missing lines
`auth.py`	44	0	100%	–
`app.py`	47	5	89%	18–20, 44, 62
Total (including tests)	164	5	97%	

The untested lines in `app.py` are the index route `/` (lines 18–20), which no test requests; the GET branch of `/login` (line 44), since the tests only send POST requests to it; and the `if __name__ == "__main__"` block (line 62), which only runs when the file is started directly. Two short integration tests would cover the first two: `GET /` when logged out and when logged in (expecting redirects to `/login` and `/welcome`), and `GET /login`. The last line is normally excluded from coverage. High coverage does not prove correctness, as 6.1 shows: the integration tests executed the password check without ever testing its boundary.

7. Conclusion

The application's separation between business logic and web layer makes both kinds of test easy to write: unit tests exercise `auth.py` without Flask, and the injectable store lets integration tests inspect the backend after an HTTP request. The fault-injection experiment showed that the two levels catch different defects, and that boundary values must be tested explicitly. The final suite has 13 passing test cases with full coverage of the business logic. The remaining gaps are two simple GET routes.

Appendix A. Added tests

```
# tests/test_unit.py
def test_hash_password_is_salted():
    first = hash_password("secret123")
    second = hash_password("secret123")

    assert first != second                      # random salt, different output
    assert verify_password("secret123", first)  # but both still verify
    assert verify_password("secret123", second)

# new row in the parametrize list of test_register_rejects_invalid_input
("a" * 21, "secret123", "Username"),            # too long (max is 20)
```

```
# tests/test_integration.py
def test_get_register_shows_form(client):
    resp = client.get("/register")

    assert resp.status_code == 200
    assert b"Create an account" in resp.data


def test_register_existing_username_fails(client, store):
    client.post("/register", data={"username": "bob", "password": "secret123"})

    resp = client.post("/register", data={"username": "bob", "password": "another123"})

    assert resp.status_code == 400
    assert b"already taken" in resp.data
    assert len(store) == 1                      # no second account saved
```