Testing a Python Web App with pytest

A hands-on guide: you'll write **3 unit tests** and **3 integration tests** for a small registration/login app.

---

1. The project

```
testing-guide/
├── app.py              # Flask web layer: routes, sessions, pages
├── auth.py             # Business logic: validation, hashing, user store
├── templates/          # HTML interface (base, form, welcome)
├── requirements.txt
└── tests/
    ├── conftest.py         # shared fixtures
    ├── test_unit.py        # 3 unit tests
    └── test_integration.py # 3 integration tests
```

Users are kept in a Python `dict` inside `UserStore`. No database, so data resets when the app restarts.

---

2. Setup and run the app locally

Requires **Python 3.10+**.

```
cd testing-guide
python -m venv .venv
source .venv/bin/activate          # Windows: .venv\Scripts\activate
pip install -r requirements.txt

python app.py
```

Open **http://127.0.0.1:5000**, create an account, log in, log out.

Try it: register with password `abc`. What error do you see? Which function in `auth.py` produced it?

Stop the server with `Ctrl+C`.

---

3. Unit vs integration

	Unit test	Integration test
Tests	One function or class	Several parts working together
Here	`auth.py` directly	HTTP requests → Flask → `auth.py` → HTML
Speed	Very fast	Fast (no real server needed)

	Unit test	Integration test
Finds	Logic bugs	Wiring bugs (routes, sessions, templates)

Every test follows **Arrange → Act → Assert**: set things up, do one thing, check the result.

---

4. Fixtures: `tests/conftest.py`

pytest loads `conftest.py` automatically. Fixtures are functions that prepare something a test needs; a test gets one just by naming it as a parameter.

```
@pytest.fixture
def store():
    return UserStore()            # fresh, empty store for every test

@pytest.fixture
def client(store):
    app = create_app(store=store, testing=True)
    with app.test_client() as client:
        yield client
```

- `store` gives each test a clean slate, so tests can't affect each other.
- `client` depends on `store`. Both point at the **same** object, so integration tests can peek into the backend.
- `test_client()` sends fake HTTP requests without starting a server.

---

5. Unit tests: `tests/test_unit.py`

Unit 1: a pure function

```
def test_hash_and_verify_password():
    stored = hash_password("secret123")

    assert "secret123" not in stored
    assert verify_password("secret123", stored)
    assert not verify_password("wrong-pass1", stored)
```

Plain `assert` is all pytest needs. On failure it shows both sides of the comparison.

Unit 2: many inputs with `parametrize`

```
@pytest.mark.parametrize("username, password, message", [
    ("ab",        "secret123",    "Username"),
    ("bad name!", "secret123",    "Username"),
    ("alice",     "short1",       "at least 8"),
```

```
    ("alice",     "nodigitshere", "digit"),
])
def test_register_rejects_invalid_input(store, username, password,
message):
    with pytest.raises(AuthError, match=message):
        store.register(username, password)
    assert len(store) == 0
```

- `parametrize` runs the test once per row (4 runs, 1 function).
- `pytest.raises` passes only if the exception is raised; `match` checks its message.

Unit 3: state and edge cases

```
def test_register_rejects_duplicate_username(store):
    store.register("Alice", "secret123")

    with pytest.raises(AuthError, match="already taken"):
        store.register("alice", "another123")

    assert len(store) == 1
    assert store.login("ALICE", "secret123")
```

Tests a business rule: usernames are case-insensitive.

---

6. Integration tests: `tests/test_integration.py`

Integration 1: the happy path

```
def test_register_then_login_shows_welcome(client, store):
    resp = client.post("/register", data={"username": "bob", "password":
"secret123"})
    assert resp.status_code == 302
    assert resp.headers["Location"].endswith("/login")
    assert store.exists("bob")

    resp = client.post("/login", data={"username": "bob", "password":
"secret123"},
                       follow_redirects=True)
    assert resp.status_code == 200
    assert b"Welcome, bob!" in resp.data
```

- `data=` sends form fields, like the browser does.
- `follow_redirects=True` lands on the final page.
- `resp.data` is **bytes**, hence `b"..."`.

Integration 2: the error path

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

    resp = client.post("/login", data={"username": "bob", "password":
"wrong-pass1"})

    assert resp.status_code == 401
    assert b"Invalid username or password" in resp.data
    with client.session_transaction() as sess:
        assert "user" not in sess
```

`session_transaction()` lets you inspect the session cookie's contents.

Integration 3: sessions across requests

```
def test_welcome_requires_login_and_logout_clears_session(client):
    resp = client.get("/welcome")
    assert resp.status_code == 302

    client.post("/register", data={"username": "bob", "password":
"secret123"})
    client.post("/login",    data={"username": "bob", "password":
"secret123"})
    assert client.get("/welcome").status_code == 200

    client.post("/logout")
    assert client.get("/welcome").status_code == 302
```

The client keeps cookies between calls, just like a browser.

---

7. Running the tests

```
pytest                          # run everything
pytest -v                       # one line per test
pytest tests/test_unit.py       # one file
pytest -k duplicate             # tests whose name matches
pytest -x                       # stop at first failure
```

Expected: `9 passed` (6 test functions; the parametrized one counts 4 times).

---

8. Exercises

1. **Break it on purpose.** In `auth.py`, change `MIN_PASSWORD_LENGTH` to `4`. Which tests fail? Revert.

2. **Unit:** add a test that `hash_password` returns a *different* string each time for the same password. Why does that matter?
3. **Unit:** add a parametrize row for a 21-character username.
4. **Integration:** test that `GET /register` returns `200` and contains `Create an account`.
5. **Integration:** test that registering an existing username returns `400` and shows `already taken`.
6. **Bonus:** install `pytest-cov` and run `pytest --cov=. --cov-report=term-missing`. Which lines are untested?