2 Testing

CS 425
Web Applications Development

Alex Petrovici

Content

Initial setup

Let's suppose a basic flask application:

```
|-- app
|   |-- flask_app.py
|   +-- main.py
|-- poetry.lock
|-- pyproject.toml
+-- tests
    +-- test_backend.py
```

Initial setup

flask_app.py

```
from flask import Flask

def create_app() -> Flask:
    app = Flask(__name__)

    @app.route("/")
    def home():
        return "Hello, Flask!"

    @app.route("/about")
    def about():
        return "This is a simple Flask application."

    return app
```

Initial setup

main.py

```python
from flask_app import create_app

if __name__ == "__main__":
    app = create_app()
    app.run(port=7000, debug=False)
```

Backend / unittest

Part of the standard python library.

```python
import unittest

from app.flask_app import create_app  # type: ignore


class FlaskAppTests(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        # Put the app in testing mode and create a test client once for all tests
        app = create_app()
        app.config.update(TESTING=True)
        cls.client = app.test_client()

    def test_home_should_return_200_and_expected_text(self):
        res = self.client.get("/")
        self.assertEqual(res.status_code, 200)
        self.assertEqual(res.mimetype, "text/html")
        self.assertIn(b"Hello, Flask", res.data)

if __name__ == "__main__":
    unittest.main(verbosity=2)
```

Backend / unittest: How to run

In the terminal you can run:

```
python -m unittest -v
```

or discover all tests under ./test

```
python -m unittest discover -s test -p "test_*.py" -v
```

Expected output:

```
test_about_should_return_200_and_expected_text
↪  (test_backend.FlaskAppTests.test_about_should_return_200_and_expected_text) ... ok
test_home_should_return_200_and_expected_text
↪  (test_backend.FlaskAppTests.test_home_should_return_200_and_expected_text) ... ok
test_unknown_route_should_return_404 (test_backend.FlaskAppTests.test_unknown_route_should_return_404)
↪  ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.006s

OK
```

Backend / pytest

External library, must be added to pyproject.toml (it's a dev dependency).

- conftest.py files are automatically discovered by pytest
- Fixtures defined in conftest.py at the project root are available to all test files in the project
- If we put it inside the tests/ folder, it would only be available to tests within that specific directory

```
|-- app
|   |-- flask_app.py
|   +-- main.py
+-- conftest.py
|-- poetry.lock
|-- pyproject.toml
+-- tests
    +-- test_backend_pytest.py
```

Backend / pytest

confest.py

```python
import pytest
from app.flask_app import create_app


@pytest.fixture
def client():
    """Create a test client for the Flask app."""
    app = create_app()
    app.config.update(TESTING=True)
    return app.test_client()
```

Backend / pytest

test_backend_pytest.py

```
class TestFlaskApp:
def test_home_should_return_200_and_expected_text(self, client):
    res = client.get("/")
    assert res.status_code == 200
    assert res.mimetype == "text/html"
    assert b"Hello, Flask" in res.data

def test_about_should_return_200_and_expected_text(self, client):
    res = client.get("/about")
    assert res.status_code == 200
    assert b"simple Flask application" in res.data

def test_unknown_route_should_return_404(self, client):
    res = client.get("/__does_not_exist__")
    assert res.status_code == 404
```

Backend / pytest: How to run

Then we can run the tests directly:

```
python3 -m pytest tests/test_backend_pytest.py -v
```

Expected output:

```
============================= test session starts ==============================
platform linux -- Python 3.13.5, pytest-8.4.2, pluggy-1.6.0
rootdir: /home/alex/projects/siu/cs425/examples/flask_app
configfile: pyproject.toml
testpaths: test
collected 3 items

tests/test_backend_pytest.py ...                                         [100%]

============================== 3 passed in 0.02s ===============================
```

or discover all tests under ./test, just run pytest from the project root

```
pytest -v
```

Backend / pytest: How to debug

To debug we have to setup the debug config to launch pytest instead of python.

`.vscode/launch.json`

```
{
    "name": "pytest: debug tests",
    "type": "debugpy",
    "request": "launch",
    "module": "pytest",
    "args": [
        "tests/test_backend_pytest.py",
        "-v",
        "-s"
    ],
    "console": "integratedTerminal",
    "cwd": "${workspaceFolder}/examples/flask_app",
    "env": {
        "PYTHONPATH": "${workspaceFolder}/examples/flask_app"
    }
}
```

Backend / pytest: Requirements when implementing pytest

Naming convention

Pytest's default discovery is:

- Files: `test_*.py` or `*_test.py`
- Functions: `def test_*()`
- Classes: `class TestSomething:  (no __init__)`
- Methods in classes: `def test_*()`

Can be overriden in pyproject.toml

`pyproject.toml`

```
[tool.pytest]
testpaths = tests
python_files = test_*.py
python_classes = Test*
python_functions = test_*
```

Frontend / Introduction

If we want to check the behaviour (JavaScript) in a webpage we need to use a program that automates a browser.

Browser Automation Frameworks

- Selenium – The most popular web automation library, supports multiple languages (Python, Java, C#, JavaScript, etc.).
- Playwright – Modern alternative from Microsoft, supports Chromium, Firefox, and WebKit with fast parallel testing.
- Puppeteer – Node.js library for controlling Chrome/Chromium; great for headless testing.
- Cypress – JavaScript-based end-to-end testing framework.
- WebDriverIO – WebDriver/Selenium-based framework for Node.js.
- TestCafe – JavaScript-based, does not require WebDriver; runs directly on browsers.

Frontend / Selenium

Automation framework

We have different frameworks available (previous slide), but we will be using Selenium because: You are invited to use Playwright since its newer, faster and has a pytest plugin.

- Mature and widely adopted
- Supports complex interactions (drag & drop), file upload

```
poetry add selenium webdriver-manager --group dev
```

Browser

We will be using Chromium because:

- Chromium is the open-source project that Chrome is built on.
- On Ubuntu, Chromium is distributed via the official repositories (easier to install).

```
sudo apt-get update && sudo apt-get install -y chromium-browser chromium-chromedriver
```

Frontend / Selenium

How to add testing. What do we want to achieve ?

`tests/test_frontend.py`

```python
from selenium.webdriver.remote.webdriver import WebDriver


def test_home_should_return_200_and_expected_text(server_url: str, driver: WebDriver) -> None:
    driver.get(f"{server_url}/")
    # We cannot directly assert status code via Selenium; content check acts as proxy
    assert "Hello, Flask!" in driver.page_source


def test_about_should_return_200_and_expected_text(server_url: str, driver: WebDriver) -> None:
    driver.get(f"{server_url}/about")
    assert "simple Flask application" in driver.page_source
```

Now let's define `server_url` and `driver`.

Frontend / Selenium

Where is chromium located ?

```
which chromium-browser
```

We should see:

```
/usr/bin/chromium-browser
```

`conftest.py`

```python
...
@pytest.fixture(scope="session")
def driver() -> Generator["WebDriver", None, None]:
    options = ChromeOptions()
    options.binary_location = "/usr/bin/chromium-browser"
    browser = webdriver.Chrome(service=Service(), options=options)
    try:
        yield browser
    finally:
        browser.quit()
```

Frontend / Selenium

conftest.py

```python
...
@pytest.fixture(scope="session")
def server_url() -> Generator[str, None, None]:
    """
    Start the Flask app in a background thread on a free port and yield its base URL.
    The thread is daemonized and will exit when the process ends.
    """
    app = create_app()
    port = _get_free_port()

    def run() -> None:
        # Werkzeug dev server; disable reloader so it doesn't spawn children
        app.run(host="127.0.0.1", port=port, debug=False, use_reloader=False)

    thread = threading.Thread(target=run, name="flask-test-server", daemon=True)
    thread.start()
    _wait_for_server("127.0.0.1", port, timeout_seconds=10.0)
    yield f"http://127.0.0.1:{port}"
    # No explicit shutdown since app has no shutdown route; daemon thread ends with process
```

Automate testing / pre-commit

1. Install dependencies

We need to install pre-commit.

```
poetry add --group dev pre-commit
```

Automate testing / pre-commit: 2. Create `.pre-commit-config.yaml`

```yaml
repos:
- repo: local
    hooks:
    - id: ruff-lint
        name: Ruff lint
        entry: poetry run ruff check --fix .
        language: system
        pass_filenames: false
```

`- repo: local` The hooks to run are defined within this repo.

`hooks:` A list of commands to run from this repository.

`entry: <entry>` The command that pre-commit executes when the hook runs.

`language: <language>` How to manage the environment that contains the tools.

To see all the options when making a new hook check the official documentation: `https://pre-commit.com/#new-hooks`

Automate testing / pre-commit:

Lets add another one.

```
...
    - id: ruff-format
      name: Ruff format
      entry: poetry run ruff format .
      language: system
      pass_filenames: false
```

All these where running with the default stages value which is `all stages` value.
Supported git hooks:

- pre-commit
- pre-merge-commit
- pre-push
- ... to see more options: https://pre-commit.com/#supported-git-hooks

Automate testing / pre-commit:

Lets add another one

```
...
    - id: pytest
        name: Run pytest
        entry: bash
        language: system
        pass_filenames: false
        stages: [pre-push]
        args:
        - -c
        - |
            set -e
            echo "Running tests with Poetry..."
            poetry run pytest -q
//
```

Automate testing / pre-commit

3. Enable the hook locally (one-time per machine)

- `poetry run pre-commit install`

  Installs the Git hook scripts into your **.git**/**hooks**/ directory.
  So that future git commit automatically triggers the hooks.

- `poetry run pre-commit install -hook-type pre-push`

  Will install the pre-push hooks instead of the default hook (pre-commit).