Leverage pytest's Fixture to write focused and highly decoupled tests

Antonio Spadaro

EuroPython 2026, 2026/07/16

Who I am

- Antonio Spadaro @ilovelinux
- DevOps Engineer @ par-tec.it
- OSS Maintainer of `datamodel-code-generator`
- Python enthusiast since Python 3.4

par-tec.it

- Software & Infrastructure system integrator
- Active in the open source scene for 20+ years
- A team of 200+ employees with 200+ professional certifications

Ever happened to you?

- One tiny refactor → **half the suite is red**
- `test_user_flow_3` ... what does it *actually* test?
- 30 lines of setup, 2 lines of test
- The same boilerplate, copy-pasted everywhere

Agenda

Unleash the power of pytest fixtures
by answering **five questions**:

1. What *is* a fixture?
2. *Where* should I put it?
3. *How* do I use it?
4. What *can* it do?
5. How do I *stop* using it?

1 / 5

What *is* a fixture?

A test suite like many others

```python
# tests/test_europython.py
from my_backend import (
    app, validate_password,
)

def test_validator_ok():
    assert validate_password("P4$$w0rd!")

def test_auth_ok():
    client = app.test_client()
    response = client.post(
        "/auth", json={"pwd": "P4$$w0rd!"}
    )

    assert response.status_code == 200
```

What's wrong here?

- The valid password is hard-coded... **twice**
- Every test builds its own `client`
- Setup noise hides the test's **intent**
- New password policy → edit *every* test

The tests know **too much** about the codebase.

Simple and straightforward

```python
# tests/conftest.py
import pytest

@pytest.fixture
def valid_password() -> str:
    return "P4$$w0rd!"
```

No import needed:
pytest matches
fixtures by **name**

```python
# tests/test_europython.py
from my_backend import (
    app, validate_password,
)

def test_validator_ok(valid_password: str):
    assert validate_password(valid_password)

def test_auth_ok(valid_password: str):
    client = app.test_client()
    response = client.post(
      "/auth", json={"pwd": valid_password}
    )

    assert response.status_code == 200
```

What just happened?

```
def test_validator_ok(valid_password: str):
```

- pytest reads each test's **parameters**
- finds the **fixture** with the same name
- calls it and **injects** the result
- Dependency Injection: no wiring, no imports

2 / 5

Where should I put it?

Three possible homes

1. **The test module**: used by one file only
2. **`conftest.py`** : shared across a directory tree
3. **A plugin**: installed with `pip`
   (`mocker` from *pytest-mock*, `client` from *pytest-django*, ...)

Rule of thumb: as close to its users as possible.

conftest.py

```
tests/
├── conftest.py          ← every test sees these
├── unit/
│   ├── conftest.py      ← unit/ only
│   └── test_validators.py
└── api/
    ├── conftest.py      ← api/ only
    └── test_auth.py
```

Auto-discovered by pytest. **Never** `import conftest`

Overriding fixtures

```python
# tests/conftest.py
import pytest

@pytest.fixture
def valid_password() -> str:
    return "P4$$w0rd!"
```

```python
# tests/legacy/conftest.py
import pytest

@pytest.fixture
def valid_password() -> str:
    # the old rules...
    return "Password!"
```

The **closest** definition wins

(it can even request the fixture it replaces)

3 / 5

How do I use it?

Just ask

```
def test_auth_ok(client, valid_password): ...
```

- Name a parameter after a fixture, **that's the whole API**
- Request as many as you need
- Requested twice in the same test?
  Created **once**. Values are cached

Keep the code focused

```python
# tests/conftest.py
import pytest
from my_backend import app

@pytest.fixture
def valid_password() -> str:
    return "P4$$w0rd!"

@pytest.fixture
def client():
    return app.test_client()
```

```python
# tests/test_europython.py

def test_auth_ok(
    client,  # ← from conftest.py
    valid_password: str,
):
    response = client.post(
        "/auth",
        json={"pwd": valid_password},
    )

    assert response.status_code == 200
```

The test file no longer imports `my_backend` at all

The policy changes...

New rule: at least **16 characters**

```
 # tests/conftest.py
 @pytest.fixture
 def valid_password() -> str:
-    return "P4$$w0rd!"
+    return "P4$$w0rd!-@EP2026!!"
```

One edit. Every test follows.

That's decoupling.

Fixtures can request fixtures

```python
# tests/conftest.py
import pytest

@pytest.fixture
def auth_headers(client, valid_password) -> dict:
    response = client.post(
        "/auth", json={"pwd": valid_password}
    )
    token = response.json["token"]
    return {"Authorization": f"Bearer {token}"}
```

```python
# tests/test_profile.py

def test_me(client, auth_headers):
    response = client.get(
        "/me", headers=auth_headers
    )

    assert response.status_code == 200
```

`client` is created **once**: test and fixture share it

Tomorrow the auth flow
switches to OAuth.
How many tests do you rewrite?

Zero.

Update `auth_headers`. Done.

usefixtures & autouse

```python
@pytest.fixture
def admin_user(client) -> None:
    client.post("/signup", json={
        "user": "root", "admin": True
    })

@pytest.mark.usefixtures("admin_user")
def test_admin_panel(client):
    res = client.get("/admin")
    assert res.status_code == 200
```

No value needed

```python
# tests/conftest.py
from my_backend import app

@pytest.fixture(autouse=True)
def app_test_mode() -> None:
    app.config["TESTING"] = True
```

every test gets it

4 / 5

What *can* it do?

Build your test data

```python
# tests/conftest.py
import secrets, string
import pytest

@pytest.fixture
def make_password():
    def _build(*, length=16, digits=True) -> str:
        chars = string.ascii_letters + "$!"
        if digits:
            chars += string.digits
        return "".join(secrets.choice(chars)
                       for _ in range(length))
    return _build
```

```python
# tests/test_europython.py

def test_rejects_short(make_password):
    pwd = make_password(length=4)
    assert not validate_password(pwd)

def test_requires_digits(make_password):
    pwd = make_password(digits=False)
    assert not validate_password(pwd)
```

Factory fixture: build only what each test needs

Same tests, more cases

```python
# tests/conftest.py
import pytest

@pytest.fixture(
    params=[
        "Sh0rt!",         # too short
        "NoDigitsHere!", # no digits
        "N0SymbolsHere", # no symbols
    ]
)
def invalid_password(
    request: pytest.FixtureRequest
) -> str:
    return request.param
```

```python
# tests/test_europython.py

def test_validator_ko(invalid_password: str):
    assert not validate_password(invalid_password)

def test_auth_ko(client, invalid_password):
    response = client.post(
      "/auth", json={"pwd": invalid_password}
    )

    assert response.status_code == 401
```

Tests, multiplied

```
$ pytest -v
test_europython.py::test_validator_ko[Sh0rt!] PASSED
test_europython.py::test_validator_ko[NoDigitsHere!] PASSED
test_europython.py::test_validator_ko[N0SymbolsHere] PASSED
test_europython.py::test_auth_ko[Sh0rt!] PASSED
test_europython.py::test_auth_ko[NoDigitsHere!] PASSED
test_europython.py::test_auth_ko[N0SymbolsHere] PASSED

================== 6 passed in 0.12s ==================
```

- 2 tests × 3 params = **6 runs**: future tests included

Pay expensive setup once

```python
@pytest.fixture(scope="session")
def database():
    return connect(start_postgres())  # ~3 s
```

scope	created
`function` *(default)*	once per **test**
`class` / `module` / `package`	once per group
`session`	once per **run**

⚠️ broader scope = shared state between tests

Enter `yield`

```python
# tests/conftest.py
@pytest.fixture
def user(database):
    user = database.add_user("antonio")  # ← setup

    yield user                           # ← test runs here

    database.remove_user(user)           # ← teardown
```

Swap `return` for `yield`:

code after it runs **when the test is done**

Teardown you can trust

- Runs **even when the test fails**
- Multiple fixtures? Torn down in **reverse order** (LIFO)
- Context managers fit right in:

```
@pytest.fixture
def client():
    with app.test_client() as client:
        yield client
```

The full lifecycle

```
# tests/conftest.py
@pytest.fixture(scope="session")
def database():
    container = start_postgres()  # before the first test
    yield connect(container)
    container.stop()              # after the last test
```

- One Postgres for the **whole run**, tests stay fast
- Nothing left running afterwards
- (that unstopped Postgres? answered)

Some fixtures come for free

fixture	you get
`tmp_path`	a fresh temporary directory
`monkeypatch`	patch attrs & env vars, auto-undone
`capsys`	captured stdout / stderr
`caplog`	captured log records

💡 **`pytest --fixtures` lists them all (plugins included)**

5 / 5

How do I *stop* using it?

So that's how fixtures stop.
But how do you *stop using them?*

You don't.

Once they click, there's no going back 😉

Recap

1. What *is* a fixture? **A dependency, injected by name**
2. *Where* should I put it? **Close to its users**
3. *How* do I use it? **Just ask. Fixtures can ask too**
4. What *can* it do? **Build, multiply, reuse, clean up**
5. How do I *stop* using it? **You don't!**

That's how you write focused, decoupled tests.

Learn more

- 🔍 `pytest --fixtures` : see what you have
- 🧩 1500+ plugins with ready-made fixtures

📖 *How to use fixtures*

Thank you!

antonio.spadaro@par-tec.it
LinkedIn: antonio-spadaro

https://ilovelinux.dev/

Slides

https://ilovelinux.dev/ep26