CODING NOMADS

11) Introduction To Testing Your App Lesson

Using Pytest Fixtures to Test Your Flask App

16 min to complete · By Brandon Gigous

Contents

In this lesson, you'll learn about how to use Pytest fixtures to create reusable chunks of test code and then use them in your Flask app.

What's A Test Fixture?

A **test fixture** is defined in the Wikipedia as:

> *A software test fixture sets up a system for the software testing process by initializing it, thereby satisfying any preconditions the system may have.*

That sounds like a mouthful, but it's simpler than you might think. Test fixtures help you do any setup that your tests may need, and even help you "tear down" your setup if necessary. As the pytest documentation puts it:

> *They provide a fixed baseline so that tests execute reliably and produce consistent, repeatable results.*

So what do these things look like and how do they work?

Pytest Fixtures

Fixtures in pytest are defined as functions, but not only that: the functions must be decorated with the `pytest.fixture` decorator. Here's a quick example:

```
@pytest.fixture
def my_list():
    return [3, 4, 7, 8]

def test_3_in_list(my_list):
    assert 3 in my_list
```

```
def test_4_in_list(my_list):
    assert 4 in my_list
```

The fixture `my_list` simply returns a list of numbers. In order to use the fixture, the `test_3_in_list()` function declares the `my_list` fixture as a parameter. Then, it simply references the fixture when checking if 3 exists in the list. You'll notice that it doesn't haven't to explicitly define the same list, it instead uses the list returned from the fixture as if it *were* defined in the `test_3_in_list()` function. The same is true in the `test_4_in_list()` function, and this is the beauty of fixtures: they can get your tests initialized so your tests don't have to do extra work.

The code looks a little counter-intuitive, doesn't it? How does the `test_3_in_list` know that the `my_list` fixture returns `[3, 4, 7, 8]` just from declaring a parameter with the same name?

Pytest is doing some magic behind the scenes. First, it finds any fixtures you defined. Then, when it discovers your test functions, it passes the value(s) returned by the fixtures with the same name.

Pytest Fixture Scope

The scope of a fixture directly relates to how often the fixture is run per test. In other words, when resources are initialized by a fixture, they *survive* for a certain period of time. Here's a table of the scope of a fixture, and what that means for how often it is run:

Scope	Frequency
`function`	Run once per test
`class`	Run once per class
`module`	Run once per module
`session`	Run once per session

Fixtures by default have a scope of `function`. In this last example, that means the `my_list` fixture was run *twice*, once per test function that used it. A list like that only

needs to be run once, so if `my_list` were set to the scope `module` it would be run once but would still be accessible to both functions.

Be sure to check out pytest's documentation on fixtures, it provides a great description and examples of how tests and fixtures actually work.

The `conftest.py` File

You just learned about the different scopes a fixture can take on. If you've been trying out some tests with the `python -m pytest` command, you've probably only had one or two files that contained all your tests. However, what if you had two files or *modules* with tests that needed to use the same fixture? Let's assume the tests above were separated into two different files, or modules:

```
# test_file_1.py
def test_3_in_list(my_list):
    assert 3 in my_list
```

```
# test_file_2.py
def test_4_in_list(my_list):
    assert 4 in my_list
```

You could set the `my_list` fixture to have a scope of `module`, where the fixture would be run twice, once per module. but would the fixture go in one of the files, both, or perhaps a different one? This is where the `conftest.py` file comes in, it's a one-stop-shop where you can place your fixtures to be accessed by *any* test file located in the same directory.

The `conftest.py` file isn't required for pytest to run tests, but it can make your life a whole lot easier. It makes it so that tests from multiple modules can access common fixtures. What might that look like, file structure-wise? The following files you don't need to make yourself, but with your test setup it might look like this:

```
tests/
├── conftest.py # my_list fixture defined here
```

```
└── unit/
    ├── __init__.py
    ├── test_file_1.py
    └── test_file_2.py
```

Then all you would need to do is call `python -m pytest tests/` as you've done before. You can read more about `conftest.py` here.

Setup *and* Teardown

While you can have a fixture that sets up resources and provides them to your test functions, you can also have it *teardown* those resources so that your test environment is in a clean state. As said before, it's important for your database to be in a clean state (all rows deleted) at the end of a test or set of tests.

So how can you have a fixture that does that? By using the `yield` keyword instead of `return`, your function can have a setup phase *and* a teardown phase. Here's a silly example to demonstrate:

```
@pytest.fixture
def order():
    # setup resource (order food)
    diner_order = DinerOrder()
    # return resource (give order to test)
    yield diner_order
    # teardown resource (remove plates, etc...)
    diner_order.clean_up()
```

- State of the tech job market – what's the deal with layoffs, AI taking our jobs, and shady coding bootcamps?
- Who IS getting jobs right now – and the skills in high demand
- What it really takes to get a technical job today – it's more than just tech skills
- And finally, is pursuing these paths still worth your time and money?

Register Here

Rethinking Database Insertion Test

Now that you've gotten the rundown of fixtures, take a look at the tests you made in previous lessons and think about how you might make a fixture that initializes your app and the test database and tears them down at the end. Try to combine all or most of the knowledge you gained from this lesson.

Learn by Doing

Task: Make a fixture `new_app` with scope `module` that:

- Creates an instance of your app.
- Pushes an application context.
- Creates the tables of the database.
- Yields an app instance.
- Tears down the database.
- Tears down the application context.

Try testing it our by passing the fixture to a new test function.

Once you give this some thought (challenge yourself!), look below to see an example of this fixture.

Reviewing your Fixture

You were tasked with creating a Pytest fixture that you could use throughout your tests. How did you do? It probably looks something like this:

```
@pytest.fixture(scope='module')
def new_app():
    # setup
    app = create_app('testing')
    assert 'data-test.sqlite' in app.config['SQLALCHEMY_DATABASE_URI
    test_client = app.test_client()
    ctx = app.app_context()
    ctx.push()
    db.create_all()

    # testing begins
    yield test_client

    # teardown
    db.session.remove()
    db.drop_all()
    ctx.pop()
```

If not, no worries, you're learning after all. :) Remember this goes in the `conftest.py` file. First is the app creation using the `testing` configuration. And instead of using the application instance directly, you use a `test_client` instead, as recommended in the Flask documentation for testing. The application context is pushed, the database tables created, and then the application is provided to whichever tests are using it. Remember that the `scope` is `module`, so all tests in a python script full of test functions that use the fixture will be given the test application instance `test_client`.

Then once the tests that use the fixture have completed, the teardown begins. First the database session is removed, then the tables are dropped. Finally, the application context is popped.

Database Insertion Test Revisited

Now that you've got this new fixture, it's time to use it! Let's take another look at that database insertion test a few lessons ago. There was a lot going on there, but now with your new fixture, it will look a lot simpler:

```
from app import db
from app.models import User

def test_database_insert(new_app):
    u = User(email='john@example.com', username='john')
    db.session.add(u)
    db.session.commit()
```

As a reminder of how fixtures work, the `test_client` yielded from the fixture is available as `new_app` in the `test_database_insert` test function. Pytest has done the voodoo behind the scenes, all you had to do was make `new_app` as a parameter to your test function. Notice it's the same name as your fixture!

But the test itself looks pretty simple, doesn't it? Indeed, the fixture has made life quite easy in this case, because your test can focus on what it needs to and lets the fixture take care of the rest. In this case, a `User` is created, then added to the database.

- Constant, personal, in-depth support 7 days a week via Discord
- Accountability, feedback, encouragement, and success

Get Mentorship

More Tests To Come

For now, you have learned the basics of using pytest to test your Flask app. Great job in keeping up! Later, as you continue in the course, you'll be tasked with creating more tests that will ensure correct functionality of the new parts of your app as you build them. A tested app is a working app!

Summary: Pytest Fixtures

In this lesson you've:

- Got acquainted with **test fixtures**, understanding that they're used to set up and tear down test environments.
- Learned about **Pytest fixtures**, defined by functions decorated with `@pytest.fixture` to create reusable test components.
- Discovered that fixtures can have different **scopes** – `function`, `class`, `module`, `session` – affecting how often they're run.
- Found out about the `conftest.py` file, which allows for sharing fixtures across various test files within the same directory.
- Saw how fixtures can handle both **setup and teardown** with the `yield` keyword, helping maintain a clean test environment.
- Created a `new_app` Pytest fixture with a `module` scope to set up and tear down your Flask app context and test database effectively.

Previous Next → Flask Security with Hashed Passwords