11) Introduction To Testing Your App Lesson

Set Up Pytest for Your Flask Project

19 min to complete · By Brandon Gigous

Contents

In this lesson, you'll learn how to use Python pytest to make your unit tests. This will be a mini crash course in how to get you up and running with pytest and Flask.

What Is Python Pytest?

Pytest is a Python framework for unit testing. What's so special about it other than that? Well, let's see what its documentation has to say.

> *Python pytest is a framework that makes building simple and scalable tests easy. Tests are expressive and readable—no boilerplate code required. Get started in minutes with a small unit test or complex functional test for your application or library.*

As was hinted at before, pytest is not just a framework, it may be the only framework you'll ever need for testing your Python code. Then again, it's also a great framework for getting started with as a beginner. Now that you have a better idea of what pytest is, next is more about how it discovers your tests.

Test Discovery

Python Pytest might be the easiest unit testing framework in existence. (DISCLAIMER: This is my exaggerated opinion.) It does most of the work of finding your tests *for you*, so all you have to worry about it is setting up your files, writing your code, and running the command to test. When you initiate the command `pytest`, it automatically looks for Python files that begin with `test_` and runs those. Within *those* files, it looks for functions that begin with `test` in their name. Couldn't be easier, right?

With that said, while you might have a basic idea for how to run a test, there's been no mention so far of how to test your *Flask app*. Let's get into that right now.

Testing Your Flask App

Next, you'll learn about how to run your first Flask app test, along with other helpful tips and additional tests.

First Test

The first step, as you probably know, is to make a file where your tests will go. Make a file called `test_app.py` and put it in your project's top-level directory, like so:

```
app/
└── ... # app package files
... # all your other files and folders
test_app.py
```

That should make it so that your `app` package can easily be imported by your test file. Let's have your first test create an `app` instance using your new application factory with the `testing` configuration:

```
from app import create_app

def test_app_creation():
    app = create_app('testing')
    assert app
```

The `assert` makes sure the `app` is defined. And barring any exceptions that may occur in the `create_app` function, your test should pass! Give it a shot:

```
# from your top-level project directory
(env) $ pytest
```

Here's what you'd expect your output to be:

```
============================= test session starts ==================
platform darwin -- Python 3.9.1, pytest-5.3.5, py-1.8.1, pluggy-0.13
rootdir: /Users/Shared/git_repos/flask-webdev-master
collected 1 item

test_app.py .
```

If it worked right away, great! You wrote your first successful test for your Flask app. Didn't work? If it's an import error, see below. Otherwise, that's okay! That's what tests are for, to help you find bugs. The problem could also be with your test file itself, usually in the form of "module not found" or "X has no attribute Y". Just remember that tests are *normally* worth the trouble of getting running since they can save you time in the long wrong, so keep trying until the test passes, or get some help from Discord or from your mentor.

If your `app` package can't be found by your test script when you run `pytest`, try instead:

```
python -m pytest
```

This is because running `pytest` by itself may not add the *current directory* `.` to the `PYTHONPATH` environment variable. `PYTHONPATH` tells python where to look for your local modules/packages (like `app/`). Using `python -m pytest` ensures this.

Throughout the rest of this section, please assume that the `python -m pytest` command is the way you should run your tests.

Once you've written your first test, there's always room for more! You'll find below a couple more tests, and there's also a discussion on keeping tests consistent across the code.

Get Mentorship

Flask Test Configuration

Before going over the tests, there's something to be aware of. In the last section, you built the `TestConfig` class. Now you can put it to use in your tests. The `TestConfig` needs the `DATABASE_TEST_URL` environment variable set, or the default value should work just fine. Make sure this gets set to a *different* value from your normal development database. Your test database can be called something like `data-test.sqlite` to distinguish it.

Additional App Creation Test

Here's an another test that also ensures app creation, and makes sure that `current_app` is available once the application context is pushed. It also checks that the `TESTING` configuration variable is set to `True`.

```
from flask import current_app

def test_current_app():
    app = create_app('testing')
    app.app_context().push()
    assert current_app
    assert current_app.config['TESTING']
```

Database Insertion Test

Now what about testing something more complicated, like an insertion into the database?

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

def test_database_insert():
    app = create_app('testing')
    assert app.config['TESTING']
    # ensure test database configured
    assert 'data-test.sqlite' in app.config['SQLALCHEMY_DATABASE_URI
    app.app_context().push()
    # create all tables if not created
    db.create_all()

    u = User(email='john@example.com', username='john')
    db.session.add(u)
    db.session.commit()

    # IMPORTANT clear database for future tests
    db.session.remove()
    db.drop_all()
```

Are you starting to see a pattern with these unit tests? :) This third test checks that the correct database is configured for testing. Once confirmed, it creates any and all tables necessary, then performs the insertion. There aren't any asserts because it is assumed that SQLAlchemy has no trouble inserting a row in the `users` table, if it is able to of course. If it isn't successful you'll know in the form of an exception.

Alert: The last part denoted as "IMPORTANT" is, well, pretty important, so don't skip over these here next couple paragraphs unless you understand. At the point right before this block, your test has *changed the state* of the database: it now has a row in a table. If you created another test that assumes the database is completely empty at the time it runs and it *isn't*...

Unexpected things might happen. So it's always good that your tests *begin* in a clean or known state, and *end* in a clean or known state. Variables like

`app` in this test are local to the function, so changing it is no issue, but the database `db`, as mentioned before, is not and might be altered throughout your tests.

Once you understand that and practice being *clean* in your tests, your ready to continue your testing journey. But perhaps you have this on your mind: that's a lot of code to do a simple test. What to do about that?

Getting Organized

Indeed, that *is* a good bit of code to write for each test. Not only does it get annoying to rewrite the same chunks of code, it's also error prone and it can get messy. Plus, you may want to organize your increasingly complex test code into different folders based on what they do, such as *unit* and *functional* tests. There must be a solution or two to these problems, right?

Of course there is! You aren't taking this course for nothing, after all. ;) In the next section, you'll get your custom test code organized in a way that allows you to write tests quickly. You'll also learn about fixtures in the next lesson which will let you write re-useable code that your tests can use as necessary.

Test Setup

This example is what you've been building on throughout this course, if you think you've missed a step or two, use the course navigation to explore the rest of the course.

While there's not a whole lot in your app to test right now, it's still important that you know where you'll put your tests in your project. Here's what your `tests/` folder setup will look like:


```
└── ...
migrations/
└── ...
tests/
├── conftest.py
```

```
    └── unit/
        ├── __init__.py
        └── test_app.py
# all your other files...
```

The `unit` folder holds all your unit test files. You can have as many files in this folder as you want or need. The `unit` folder has an `__init__.py` file to make it package, which is generally a good practice when using multiple test folders with pytest. From Pytest official documentation:

> *If you need to have test modules with the same name, you might add `__init__.py` files to your tests folder and sub-folders, changing them to packages.*

Also notice that your `test_app.py` file will be moved to the `unit/` folder.

Then the `conftest.py` file is the place to *configure* your test setup with things like test fixtures. You'll learn more about `conftest.py` in a bit; for now it's fine if it's empty. Let's first talk about how you'll start your tests with this new file structure.

Starting Your Tests

While it takes little effort for Python pytest to do its magic, it still needs a little help sometimes. But now you will have all your test code in one folder called `tests/`. In a way it makes things easier, but there's still a bit of a challenge. Because of your file structure with tests in the top-level `tests/` folder and `app/` containing most of your app's logic, it takes some up-front configuration to get pytest working properly.

Generally, all you need to do in this case is to give pytest the name of the directory where the tests are, e.g. `tests/`, from the top level directory:

```
(env) $ python -m pytest tests/
```

VS Code Launch Configuration

If you're using VSCode, you can use this as a launch configuration for running tests:

```
{
  "name": "Python: Test All",
  "type": "python",
  "request": "launch",
  "module": "pytest",
  "env": {
    "DATABASE_TEST_URL": "sqlite:///${workspaceFolder}/data-test.sql
    "PYTHONPATH": ". tests"
  },
  "args": ["${workspaceFolder}/tests/", "--show-capture=stdout"]
}
```

The `${workspaceFolder}` is a variable set by VSCode that holds the location of folder you're working in. If you have your project folder open in VSCode, then this is the location of your project folder. So the `DATABASE_TEST_URL` environment variable is set to the location of your `data-test.sqlite` file. If the file doesn't exist yet, it will be created when your test database's tables are first created.

You'll notice that `PYTHONPATH` is set here as well. This sets it to current directory and additionally your `tests` directory.

You should be good to launch your tests if you've made it this far. In the next lesson, you'll learn about how to use fixtures to create reusable chunks of test code.

Summary: Python Pytest

You've successfully learned how to set up Python Pytest with Flask and started on the path of testing your application. You've:

- Used **Pytest**, a powerful testing framework.
- Understood the concept of **test discovery** where Pytest automatically identifies test files and functions by their naming pattern.
- Created your first Flask test in a file named `test_app.py` that verifies the creation of a Flask application *instance*.

- Learned about the importance of switching to a **test database** to avoid interfering with your production data, using the `TestConfig` you set up earlier in the course.
- Emphasized the need to **clean up after your tests** to prevent state changes in the database from affecting other tests.
- Discovered how to set up a **VS Code launch configuration** to streamline the process.

Happy testing!

Previous Next → Using Pytest Fixtures to Test Your Flask App