

# 8

# Testing FastAPI Applications

In the last chapter, we learned how to secure a FastAPI application using OAuth and **JSON Web Token (JWT)**. We successfully implemented an authentication system and learned what dependency injection is all about. We also learned how to inject dependencies into our routes to restrict unauthorized access and operations. We have successfully built a secure web API that has database support and is able to perform CRUD operations easily. In this chapter, we will learn what testing is and how to write tests to ensure that our application behaves as expected.

Testing is an integral part of the application development cycle. Application testing is done to ensure the correct functioning state of the application and easily detect anomalies in the application before deploying to production. Although we have been manually testing our application's endpoint in the last few chapters, we will be learning how to automate these tests.

By the end of this chapter, you will be able to write tests for your FastAPI application routes. This chapter will explain what unit testing is and how to perform unit testing on the application routes. In this chapter, you'll be covering the following topics:

- Unit testing with `pytest`
- Setting up our test environment

- Writing tests for REST API endpoints
- Test coverage

## Technical requirements

For this chapter, you'll need a running MongoDB server on your local machine. The steps outlined in *Chapter 6, Connecting to a Database*, should be followed to get your database server up and running.

The code used in this chapter can be found at <https://github.com/PacktPublishing/Building-Python-Web-APIs-with-FastAPI/tree/main/ch08/planner>.

## Unit testing with pytest

Unit testing is a testing procedure where individual components of an application are tested. This form of testing enables us to verify the working capability of individual components. For example, unit tests are employed in testing individual routes in an application to ensure the proper responses are returned.

In this chapter, we'll be making use of `pytest`, a Python testing library, to conduct our unit testing operations. Although Python comes with a unit testing library called `unittest` built in, the `pytest` library has a shorter syntax and is more preferred for testing applications. Let's install `pytest` and write our first sample test.

Let's install the `pytest` library:

```
(venv)$ pip install pytest
```

Next, create a folder called `tests` that will house the test files for our application:

```
(venv)$ mkdir tests && cd  
(venv)$ touch __init__.py
```

Individual test filenames, during creation, will be prefixed with `test_`. This will enable the `pytest` library to recognize and run the test file. Let's create a test file in the newly created `tests` directory that checks the correctness of the addition, subtraction, multiplication, and division arithmetic operations:

```
(venv)$ touch test_arithmetic_operations.py
```

Let's define the function that performs the arithmetic operations first. In the `tests` file, add the following:

```
def add(a: int , b: int) -> int:
    return a + b

def subtract(a: int, b: int) -> int:
    return b - a

def multiply(a: int, b: int) -> int:
    return a * b

def divide(a: int, b: int) -> int:
    return b // a
```

Now that we have defined the operations to be tested, we'll create the functions that'll handle these tests. In the test functions, the operation to be executed is defined. The `assert` keyword is used to verify that the output on the left-hand side is in correspondence to the output of the operation on the right-hand side. In our case, we'll be testing that the arithmetic operations equal their respective results.

Add the following to the `tests` file:

```
def test_add() -> None:
    assert add(1, 1) == 2

def test_subtract() -> None:
    assert subtract(2, 5) == 3

def test_multiply() -> None:
    assert multiply(10, 10) == 100

def test_divide() -> None:
    assert divide(25, 100) == 4
```

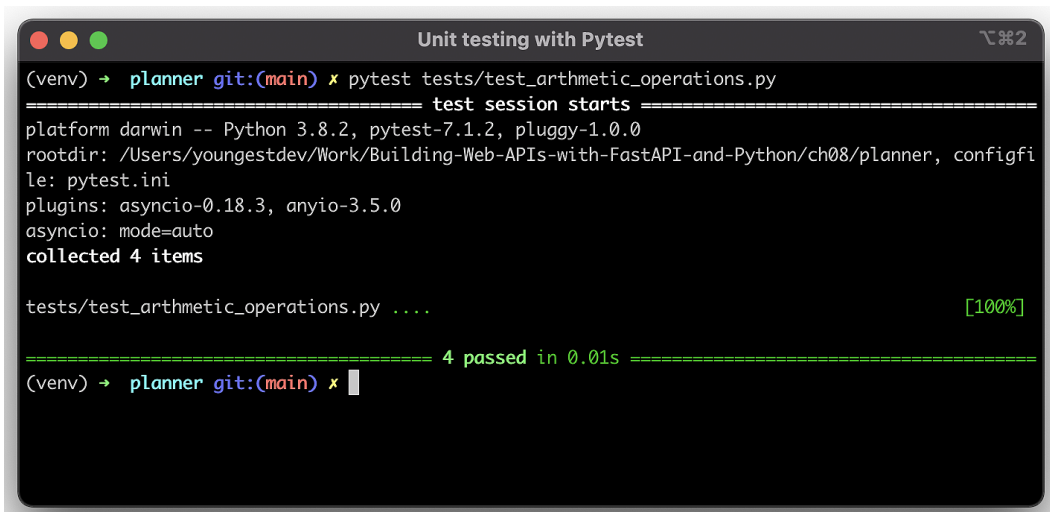
**Tip**

The standard practice is to define the functions that will be tested in an external location (`add()`, `subtract()`, and so on, in our case). This file is then imported and the functions to be tested are invoked in the test functions.

With the test functions in place, we're set to run the test file. The tests can be executed by running the `pytest` command. However, this command runs all test files contained in the folder. To execute a single test, the test filename is passed as an argument. Let's run the test file:

```
(venv)$ pytest test_arithmetic_operations.py
```

The tests defined all passed. This is signified by the response in green:



```

Unit testing with Pytest
(venv) → planner git:(main) ✕ pytest tests/test_arithmetic_operations.py
===== test session starts =====
platform darwin -- Python 3.8.2, pytest-7.1.2, pluggy-1.0.0
rootdir: /Users/youngestdev/Work/Building-Web-APIs-with-FastAPI-and-Python/ch08/planner, configfile: pytest.ini
plugins: asyncio-0.18.3, anyio-3.5.0
asyncio: mode=auto
collected 4 items

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

===== 4 passed in 0.01s =====
(venv) → planner git:(main) ✕

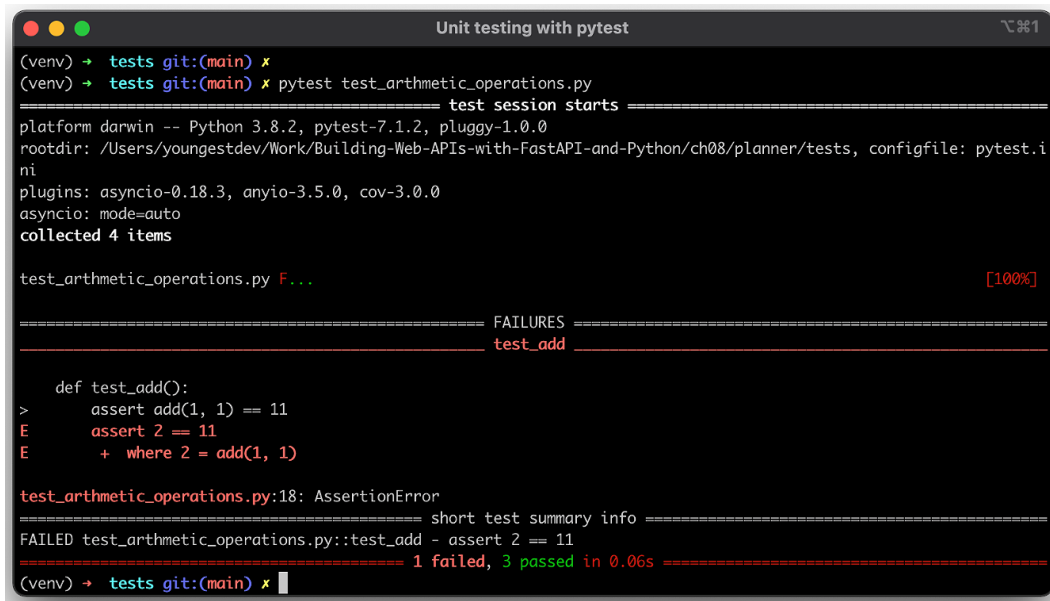
```

Figure 8.1 – Unit testing result carried out on arithmetic operations

Tests that failed, as well as the point of the failure, are highlighted in red. For example, say we modify the `test_add()` function as such:

```
def test_add() -> None:
    assert add(1, 1) == 11
```

In the following figure, the failing test, as well as the point of failure, is highlighted in red.



```
(venv) → tests git:(main) ✗
(venv) → tests git:(main) ✗ pytest test_arithmetic_operations.py

===== test session starts =====
platform darwin -- Python 3.8.2, pytest-7.1.2, pluggy-1.0.0
rootdir: /Users/youngestdev/Work/Building-Web-APIs-with-FastAPI-and-Python/ch08/planner/tests, configfile: pytest.ini
plugins: asyncio-0.18.3, anyio-3.5.0, cov-3.0.0
asyncio: mode=auto
collected 4 items

test_arithmetic_operations.py F... [100%]

===== FAILURES =====
test_add

  def test_add():
>     assert add(1, 1) == 11
E     assert 2 == 11
E     + where 2 = add(1, 1)

test_arithmetic_operations.py:18: AssertionError
===== short test summary info =====
FAILED test_arithmetic_operations.py::test_add - assert 2 == 11
===== 1 failed, 3 passed in 0.06s =====
(venv) → tests git:(main) ✗
```

Figure 8.2 – Failing test

The test failed at the `assert` statement, where the correct result, `2`, is displayed.

The failure is summarized as `AssertionError`, which tells us the test failed due to an incorrect assertion (`2 == 1`) being passed.

Now that we have an idea of how `pytest` works, let's take a look at fixtures in `pytest`.

## Eliminating repetition with `pytest` fixtures

Fixtures are reusable functions defined to return the data needed in test functions. Fixtures are decorated with the `pytest.fixture` decorator. An example use case of a fixture is returning an applications instance to execute tests for the API endpoints. A fixture can be used to define the application client that is returned and used in test functions, eliminating the need to redefine the application instance in every test. We shall see how this is used in the *Writing tests for REST API endpoints* section.

Let's look at an example:

```
import pytest

from models.events import EventUpdate
```