

```
# Fixture is defined.
@pytest.fixture
def event() -> EventUpdate:
    return EventUpdate(
        title="FastAPI Book Launch",
        image="https://packt.com/fastapi.png",
        description="We will be discussing the contents of
        the FastAPI book in this event.Ensure to come with
        your own copy to win gifts!",
        tags=["python", "fastapi", "book", "launch"],
        location="Google Meet"
    )

def test_event_name(event: EventUpdate) -> None:
    assert event.title == "FastAPI Book Launch"
```

In the preceding code block, we've defined a fixture that returns an instance of the `EventUpdate` pydantic model. This fixture is passed as an argument in the `test_event_name` function, enabling the properties to become accessible.

The fixture decorator can optionally take arguments. One of these arguments is `scope` – the scope of a fixture tells `pytest` what the duration of the fixture function will be.

In this chapter, we'll make use of two scopes:

- `session`: This scope tells `pytest` to instantiate the function once for the whole testing session.
- `module`: This scope instructs `pytest` to execute the affixed function only once the test file is executed.

Now that we know what a fixture is, let's set up our test environment in the next section.

Setting up our test environment

In the previous section, we learned the basics of testing as well as what fixtures are.

We will now test the endpoints for CRUD operations as well as user authentication. To test our asynchronous APIs, we'll be making use of `httpx` and installing the `pytest-asyncio` library to enable us to test our asynchronous API.

Install the additional libraries:

```
(venv)$ pip install httpx pytest-asyncio
```

Next, we'll create a configuration file called `pytest.ini`. Add the following code to it:

```
[pytest]
asyncio_mode = True
```

The configuration file is read when `pytest` is run. This automatically makes `pytest` run all tests in asynchronous mode.

With the configuration file in place, let's create the umbrella test file, `conftest.py`, which will be responsible for creating an instance of our application required by the test files. In the `tests` folder, create the `conftest` file:

```
(venv)$ touch tests/conftest.py
```

We'll start by importing the dependencies needed into `conftest.py`:

```
import asyncio
import httpx
import pytest

from main import app
from database.connection import Settings
from models.events import Event
from models.users import User
```

In the preceding code block, we have imported the `asyncio`, `httpx`, and `pytest` modules. The `asyncio` module will be used to create an active loop session to ensure the tests run on a single thread to avoid conflicts. The `httpx` test will act as the asynchronous client for conducting HTTP CRUD operations. The `pytest` library is needed for defining fixtures.

We have also imported our application's instance `app`, as well as the models and the `Settings` class. Let's define the loop session fixture:

```
@pytest.fixture(scope="session")
def event_loop():
    loop = asyncio.get_event_loop()
    yield loop
    loop.close()
```

With that in place, let's create a new database instance from the `Settings` class:

```
async def init_db():
    test_settings = Settings()
    test_settings.DATABASE_URL =
        "mongodb://localhost:27017/testdb"

    await test_settings.initialize_database()
```

In the preceding code block, we have defined a new `DATABASE_URL`, as well as invoking the initialization function defined in *Chapter 6, Connecting to a Database*. We're now making use of a new database, `testdb`.

Lastly, let's define the default client fixture, which returns an instance of our application run asynchronously through `httpx`:

```
@pytest.fixture(scope="session")
async def default_client():
    await init_db()
    async with httpx.AsyncClient(app=app,
        base_url="http://app") as client:
        yield client
    # Clean up resources
    await Event.find_all().delete()
    await User.find_all().delete()
```

In the preceding code block, the database is initialized first, and the application is spun as an `AsyncClient`, which is kept alive until the end of the test session. At the end of the testing session, the event and user collection are wiped off to ensure the database is empty before each test run.

In this section, you have been introduced to the steps involved in setting up your test environment. In the next section, you'll be taken through the process of writing tests for the each endpoint created in the application.

Writing tests for REST API endpoints

With everything in place, let's create the `test_login.py` file, where we'll test the authentication routes:

```
(venv)$ touch tests/test_login.py
```

In the test file, we'll start by importing the dependencies:

```
import httpx
import pytest
```

Testing the sign-up route

The first endpoint we'll be testing is the sign-up endpoint. We'll be adding the `pytest.mark.asyncio` decorator, which informs `pytest` to treat this as an async test. Let's define the function and the request payload:

```
@pytest.mark.asyncio
async def test_sign_new_user(default_client: httpx.AsyncClient)
-> None:
    payload = {
        "email": "testuser@packt.com",
        "password": "testpassword",
    }
```

Let's define the request header and expected response:

```
headers = {
    "accept": "application/json",
    "Content-Type": "application/json"
}

test_response = {
    "message": "User created successfully"
}
```

Now that we have defined the expected response for this request, let's initiate the request:

```
response = await default_client.post("/user/signup",
    json=payload, headers=headers)
```

Next, we'll test whether the request was successful by comparing the responses:

```
assert response.status_code == 200
assert response.json() == test_response
```

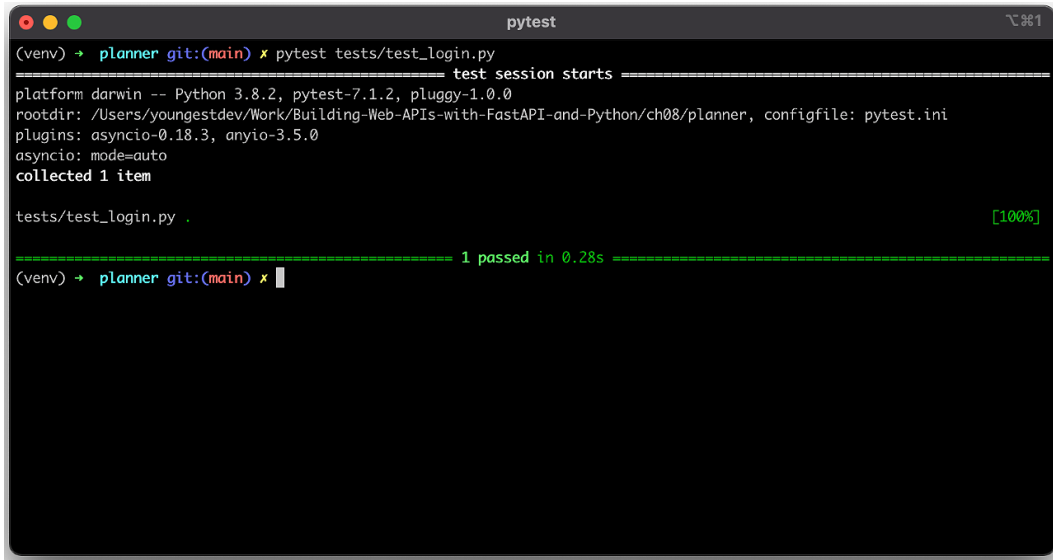
Before running this test, let's briefly comment out the line that erases user data in `conf/test.py` as it will cause the authenticated tests to fail:

```
# await User.find_all().delete()
```

From your terminal, start your MongoDB server and run the test:

```
(venv)$ pytest tests/test_login.py
```

The sign-up route has been successfully tested:



```

(pytest)
(venv) → planner git:(main) ✗ pytest tests/test_login.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 1 item

tests/test_login.py . [100%]

===== 1 passed in 0.28s =====
(venv) → planner git:(main) ✗

```

Figure 8.3 – Successful test run on the sign-up route

Let's proceed to write the test for the sign-in route. In the meantime, you can quickly tweak the test response to see whether your test fails or not!

Testing the sign-in route

Below the test for the sign-up route, let's define the test for the sign-in route. We'll start by defining the request payload and the headers before initiating the request like the first test:

```
@pytest.mark.asyncio
async def test_sign_user_in(default_client: httpx.AsyncClient)
-> None:
    payload = {
        "username": "testuser@packt.com",
        "password": "testpassword"
```