

```

    }

    headers = {
        "accept": "application/json",
        "Content-Type": "application/x-www-form-urlencoded"
    }

```

Next, we'll initiate the request and test the responses:

```

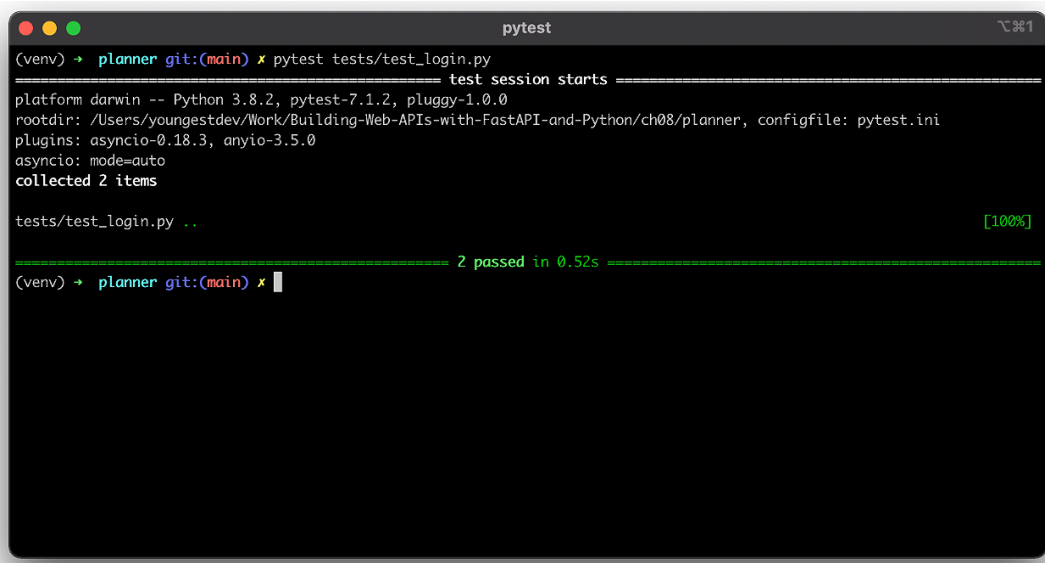
response = await default_client.post("/user/signin",
data=payload, headers=headers)

assert response.status_code == 200
assert response.json()["token_type"] == "Bearer"

```

Let's rerun the test:

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



The screenshot shows a terminal window titled 'pytest' with a dark background. The command '(venv) → planner git:(main) ✗ pytest tests/test_login.py' has been executed. The output shows the test session starting with platform 'darwin', Python version '3.8.2', and pytest version '7.1.2'. It lists the root directory as '/Users/youngestdev/Work/Building-Web-APIs-with-FastAPI-and-Python/ch08/planner' and the configfile as 'pytest.ini'. It also shows the plugins 'asyncio-0.18.3' and 'anyio-3.5.0', and that 'asyncio: mode=auto' is set. Two items were collected from 'tests/test_login.py'. The test results show '2 passed in 0.52s' with a green progress bar at the bottom. The prompt '(venv) → planner git:(main) ✗' is visible at the bottom of the terminal.

```

(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 2 items

tests/test_login.py .. [100%]

===== 2 passed in 0.52s =====
(venv) → planner git:(main) ✗

```

Figure 8.4 – Successful test run for both routes

Let's change the username for signing in to a wrong one to confirm that the test will fail:

```
payload = {
    "username": "wronguser@packt.com",
    "password": "testpassword"
}
```

```
default_client = <httpx.AsyncClient object at 0x1044a84f0>

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

    headers = {
        "accept": "application/json",
        "Content-Type": "application/x-www-form-urlencoded"
    }

    response = await default_client.post("/user/signin", data=payload, headers=headers)

> assert response.status_code == 200
E   assert 404 == 200
E   + where 404 = <Response [404 Not Found]>.status_code

tests/test_login.py:41: AssertionError

===== short test summary info =====
FAILED tests/test_login.py::test_sign_user_in - assert 404 == 200
===== 1 failed, 1 passed in 0.34s =====
(venv) + planner git:(main) x
```

Figure 8.5 – Failing test due to wrong request payload

We have successfully written tests for the sign-up and sign-in routes. Let's proceed to test the CRUD routes for the event planner API.

Testing CRUD endpoints

We'll start by creating a new file called `test_routes.py`:

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

In the newly created file, add the following code:

```
import httpx
import pytest

from auth.jwt_handler import create_access_token
from models.events import Event
```

In the preceding code block, we have imported the regular dependencies. We've also imported the `create_access_token (user)` function and the `Event` model. Since some of the routes are protected, we'll be generating an access token ourselves. Let's create a new fixture that returns an access token when invoked. The fixture has a scope of `module`, which means it is run only once – when the test file is executed – and isn't invoked on every function call. Add the following code:

```
@pytest.fixture(scope="module")
async def access_token() -> str:
    return create_access_token("testuser@packt.com")
```

Let's create a new fixture that adds an event to the database. This action is performed to run preliminary tests before testing the CRUD endpoints. Add the following code:

```
@pytest.fixture(scope="module")
async def mock_event() -> Event:
    new_event = Event(
        creator="testuser@packt.com",
        title="FastAPI Book Launch",
        image="https://linktomyimage.com/image.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"
    )

    await Event.insert_one(new_event)

    yield new_event
```

Testing READ endpoints

Next, let's write a test function that tests the **GET HTTP** method on the `/event` route:

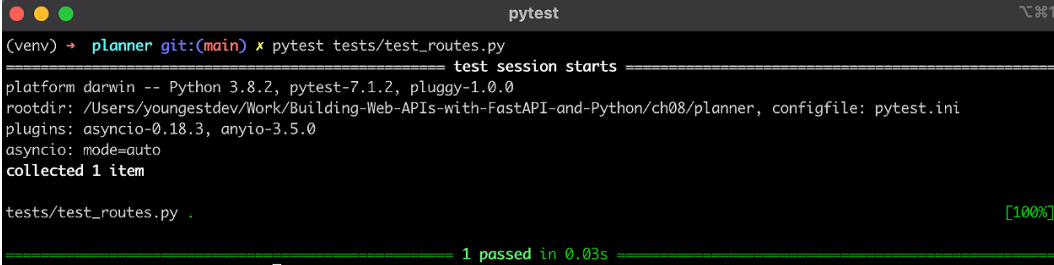
```
@pytest.mark.asyncio
async def test_get_events(default_client: httpx.AsyncClient,
mock_event: Event) -> None:
    response = await default_client.get("/event/")
```

```
assert response.status_code == 200
assert response.json()[0]["_id"] == str(mock_event.id)
```

In the preceding code block, we're testing the event route path to check whether the event added to the database in the `mock_event` fixture is present. Let's run the test:

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

Here's the result:



```
pytest
(venv) -> planner git:(main) x pytest tests/test_routes.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_routes.py . [100%]

===== 1 passed in 0.03s =====
```

Figure 8.6 – Successful test run

Next, let's write the test function for the `/event/{id}` endpoint:

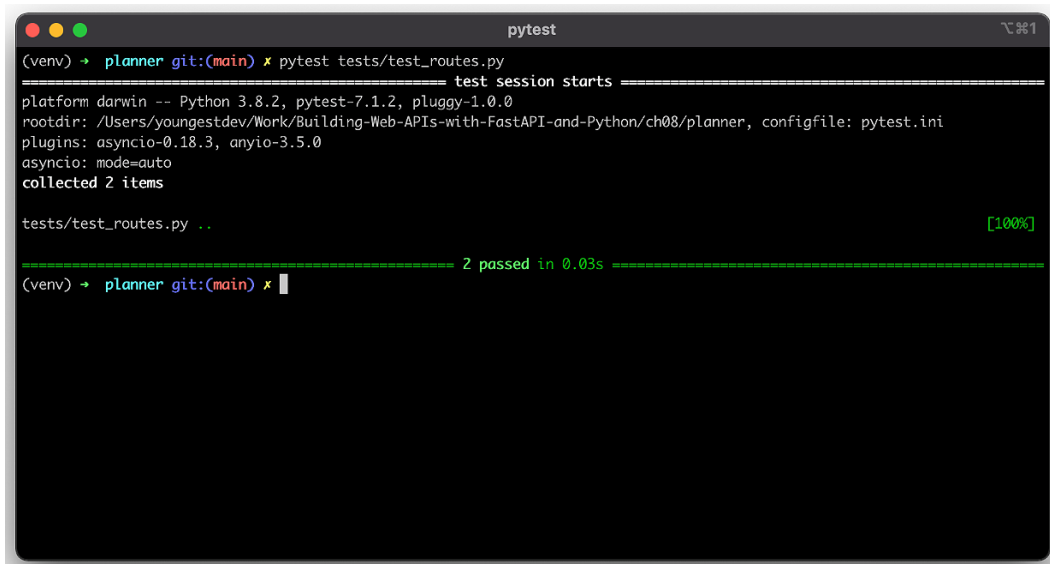
```
@pytest.mark.asyncio
async def test_get_event(default_client: httpx.AsyncClient,
mock_event: Event) -> None:
    url = f"/event/{str(mock_event.id)}"
    response = await default_client.get(url)

    assert response.status_code == 200
    assert response.json()["creator"] == mock_event.creator
    assert response.json()[ "_id"] == str(mock_event.id)
```

In the preceding code block, we're testing the endpoint that retrieves a single event. The event ID passed is retrieved from the `mock_event` fixture and the result from the request compared with the data stored in the `mock_event` fixture. Let's run the test:

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

Here's the result:



```

(pyenv) → planner git:(main) ✗ pytest tests/test_routes.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 2 items

tests/test_routes.py .. [100%]

===== 2 passed in 0.03s =====
(pyenv) → planner git:(main) ✗

```

Figure 8.7 – Successful test run for single event retrieval

Next, let's write the test function for creating a new event.

Testing the CREATE endpoint

We'll start by defining the function and retrieving an access token from the fixture created earlier. We'll create the request payload, which will be sent to the server, the request headers, which will comprise the content type, as well as the authorization header value. The test response will also be defined, after which the request is initiated and the responses compared. Add the following code:

```

@pytest.mark.asyncio
async def test_post_event(default_client: httpx.AsyncClient,
access_token: str) -> None:
    payload = {
        "title": "FastAPI Book Launch",
        "image": "https://linktomyimage.com/image.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": [

```