Async Tests

You have already seen how to test your **FastAPI** applications using the provided `TestClient`, but with it, you can't test or run any other `async` function in your (synchronous) pytest functions.

Being able to use asynchronous functions in your tests could be useful, for example, when you're querying your database asynchronously. Imagine you want to test sending requests to your FastAPI application and then verify that your backend successfully wrote the correct data in the database, while using an async database library.

Let's look at how we can make that work.

pytest.mark.anyio

If we want to call asynchronous functions in our tests, our test functions have to be asynchronous. Anyio provides a neat plugin for this, that allows us to specify that some test functions are to be called asynchronously.

HTTPX

Even if your **FastAPI** application uses normal `def` functions instead of `async def`, it is still an `async` application underneath.

The `TestClient` does some magic inside to call the asynchronous FastAPI application in your normal `def` test functions, using standard pytest. But that magic doesn't work anymore when we're using it inside asynchronous functions. By running our tests asynchronously, we can no longer use the `TestClient` inside our test functions.

Luckily there's a nice alternative, called HTTPX.

HTTPX is an HTTP client for Python 3 that allows us to query our FastAPI application similarly to how we did it with the `TestClient`.

If you're familiar with the Requests library, you'll find that the API of HTTPX is almost identical.

The important difference for us is that with HTTPX we are not limited to synchronous, but can also make asynchronous requests.

Example

For a simple example, let's consider the following `main.py` module:

`{!../../../docs_src/async_tests/main.py!}`

The `test_main.py` module that contains the tests for `main.py` could look like this now:

{!../../../docs_src/async_tests/test_main.py!}

Run it

You can run your tests as usual via:

```
$ pytest

---> 100%
```

In Detail

The marker `@pytest.mark.anyio` tells pytest that this test function should be called asynchronously:

Python hl_lines="7" {!../../../docs_src/async_tests/test_main.py!}

!!! tip Note that the test function is now `async def` instead of just `def` as before when using the `TestClient`.

Then we can create an `AsyncClient` with the app, and send async requests to it, using `await`.

Python hl_lines="9-10" {!../../../docs_src/async_tests/test_main.py!}

This is the equivalent to:

```
response = client.get('/')
```

that we used to make our requests with the `TestClient`.

!!! tip Note that we're using async/await with the new `AsyncClient` - the request is asynchronous.

Other Asynchronous Function Calls

As the testing function is now asynchronous, you can now also call (and `await`) other `async` functions apart from sending requests to your FastAPI application in your tests, exactly as you would call them anywhere else in your code.

!!! tip If you encounter a `RuntimeError: Task attached to a different loop` when integrating asynchronous function calls in your tests (e.g. when using MongoDB's MotorClient) Remember to instantiate objects that need an event loop only within async functions, e.g. an `'@app.on_event("startup")` callback.