Testing

Thanks to Starlette, testing **FastAPI** applications is easy and enjoyable.

It is based on Requests, so it's very familiar and intuitive.

With it, you can use pytest directly with **FastAPI**.

Using `TestClient`

Import `TestClient`.

Create a `TestClient` passing to it your **FastAPI** application.

Create functions with a name that starts with `test_` (this is standard `pytest` conventions).

Use the `TestClient` object the same way as you do with `requests`.

Write simple `assert` statements with the standard Python expressions that you need to check (again, standard `pytest`).

```
Python hl_lines="2  12  15-18" {!../../../docs_src/app_testing/tutorial001.py!}
```

!!! tip Notice that the testing functions are normal `def`, not `async def`.

```
And the calls to the client are also normal calls, not using `await`.

This allows you to use `pytest` directly without complications.
```

!!! note "Technical Details" You could also use `from starlette.testclient import TestClient`.

```
**FastAPI** provides the same `starlette.testclient` as `fastapi.testclient` just as a conve
```

!!! tip If you want to call `async` functions in your tests apart from sending requests to your FastAPI application (e.g. asynchronous database functions), have a look at the Async Tests in the advanced tutorial.

Separating tests

In a real application, you probably would have your tests in a different file.

And your **FastAPI** application might also be composed of several files/modules, etc.

FastAPI app file

Let's say you have a file `main.py` with your **FastAPI** app:

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

Testing file

Then you could have a file `test_main.py` with your tests, and import your `app` from the `main` module (`main.py`):

`{!../../../docs_src/app_testing/test_main.py!}`

Testing: extended example

Now let's extend this example and add more details to see how to test different parts.

Extended FastAPI app file

Let's say that now the file `main.py` with your **FastAPI** app has some other **path operations**.

It has a `GET` operation that could return an error.

It has a `POST` operation that could return several errors.

Both *path operations* require an `X-Token` header.

=== "Python 3.6 and above"

```Python
{!> ../../../docs_src/app_testing/app_b/main.py!}
```

=== "Python 3.10 and above"

```Python
{!> ../../../docs_src/app_testing/app_b_py310/main.py!}
```

Extended testing file

You could then update `test_main.py` with the extended tests:

`{!> ../../../docs_src/app_testing/app_b/test_main.py!}`

Whenever you need the client to pass information in the request and you don't know how to, you can search (Google) how to do it in `requests`.

Then you just do the same in your tests.

E.g.:

- To pass a *path* or *query* parameter, add it to the URL itself.
- To pass a JSON body, pass a Python object (e.g. a `dict`) to the parameter `json`.
- If you need to send *Form Data* instead of JSON, use the `data` parameter instead.

- To pass *headers*, use a `dict` in the `headers` parameter.
- For *cookies*, a `dict` in the `cookies` parameter.

For more information about how to pass data to the backend (using `requests` or the `TestClient`) check the Requests documentation.

!!! info Note that the `TestClient` receives data that can be converted to JSON, not Pydantic models.

`If you have a Pydantic model in your test and you want to send its data to the application c`

Run it

After that, you just need to install `pytest`:

```
$ pip install pytest

---> 100%
```

It will detect the files and tests automatically, execute them, and report the results back to you.

Run the tests with:

```
$ pytest

================ test session starts ================
platform linux -- Python 3.6.9, pytest-5.3.5, py-1.8.1, pluggy-0.13.1
rootdir: /home/user/code/superawesome-cli/app
plugins: forked-1.1.3, xdist-1.31.0, cov-2.8.1
collected 6 items

---> 100%

test_main.py <span style="color: green; white-space: pre;">......

<span style="color: green;">================= 1 passed in 0.03s =================</span>
```