Security - First Steps

Let's imagine that you have your **backend** API in some domain.

And you have a **frontend** in another domain or in a different path of the same domain (or in a mobile application).

And you want to have a way for the frontend to authenticate with the backend, using a **username** and **password**.

We can use **OAuth2** to build that with **FastAPI**.

But let's save you the time of reading the full long specification just to find those little pieces of information you need.

Let's use the tools provided by **FastAPI** to handle security.

How it looks

Let's first just use the code and see how it works, and then we'll come back to understand what's happening.

Create `main.py`

Copy the example in a file `main.py`:

Python 3.9+

```
from typing import Annotated

from fastapi import Depends, FastAPI
from fastapi.security import OAuth2PasswordBearer

app = FastAPI()

oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")


@app.get("/items/")
async def read_items(token: Annotated[str, Depends(oauth2_scheme)]):
    return {"token": token}
```

▸ Other versions and variants

Python 3.8+

```
from fastapi import Depends, FastAPI
from fastapi.security import OAuth2PasswordBearer
from typing_extensions import Annotated

app = FastAPI()

oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")


@app.get("/items/")
async def read_items(token: Annotated[str, Depends(oauth2_scheme)]):
    return {"token": token}
```

Python 3.8+ - non-Annotated

Tip

Prefer to use the `Annotated` version if possible.

```
from fastapi import Depends, FastAPI
from fastapi.security import OAuth2PasswordBearer

app = FastAPI()

oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")


@app.get("/items/")
async def read_items(token: str = Depends(oauth2_scheme)):
    return {"token": token}
```

Run it

Info

The `python-multipart` package is automatically installed with **FastAPI** when you run the `pip install "fastapi[standard]"` command.

However, if you use the `pip install fastapi` command, the `python-multipart` package is not included by default.

To install it manually, make sure you create a virtual environment, activate it, and then install it with:

```
$ pip install python-multipart
```

This is because **OAuth2** uses "form data" for sending the `username` and `password`.

Run the example with:

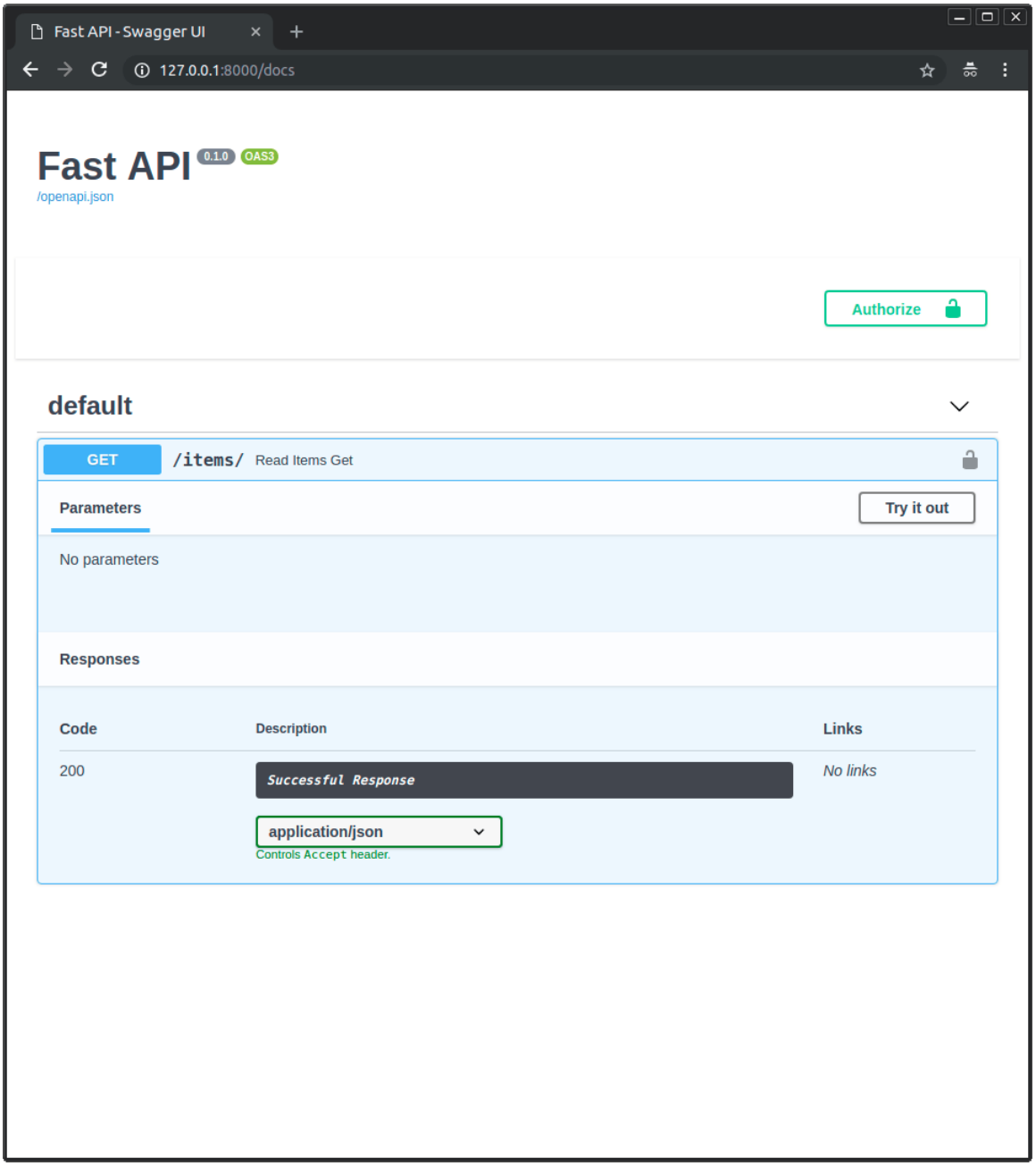
```
$ fastapi dev main.py

<span style="color: green;">INFO</span>:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
```

Check it

Go to the interactive docs at: http://127.0.0.1:8000/docs.

You will see something like this:

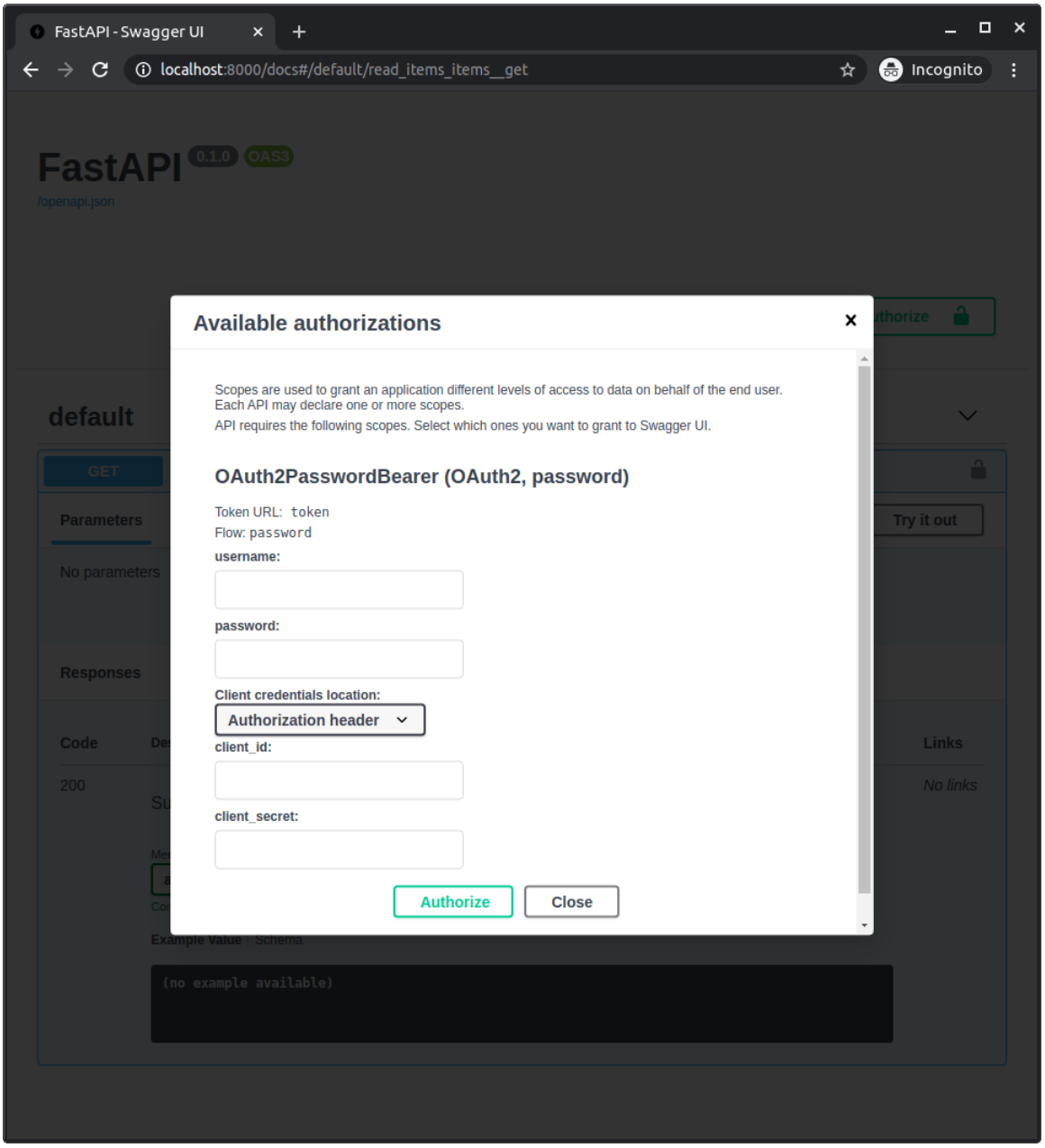


Authorize button!

You already have a shiny new "Authorize" button.

And your *path operation* has a little lock in the top-right corner that you can click.

And if you click it, you have a little authorization form to type a `username` and `password` (and other optional fields):



Note

It doesn't matter what you type in the form, it won't work yet. But we'll get there.

This is of course not the frontend for the final users, but it's a great automatic tool to document interactively all your API.

It can be used by the frontend team (that can also be yourself).

It can be used by third party applications and systems.

And it can also be used by yourself, to debug, check and test the same application.

The `password` flow

Now let's go back a bit and understand what is all that.

The `password` "flow" is one of the ways ("flows") defined in OAuth2, to handle security and authentication.

OAuth2 was designed so that the backend or API could be independent of the server that authenticates the user.

But in this case, the same **FastAPI** application will handle the API and the authentication.

So, let's review it from that simplified point of view:

- The user types the `username` and `password` in the frontend, and hits `Enter`.
- The frontend (running in the user's browser) sends that `username` and `password` to a specific URL in our API (declared with `tokenUrl="token"`).
- The API checks that `username` and `password`, and responds with a "token" (we haven't implemented any of this yet).
  - A "token" is just a string with some content that we can use later to verify this user.
  - Normally, a token is set to expire after some time.

    - So, the user will have to log in again at some point later.
    - And if the token is stolen, the risk is less. It is not like a permanent key that will work forever (in most of the cases).
- The frontend stores that token temporarily somewhere.
- The user clicks in the frontend to go to another section of the frontend web app.
- The frontend needs to fetch some more data from the API.
    - But it needs authentication for that specific endpoint.
    - So, to authenticate with our API, it sends a header `Authorization` with a value of `Bearer` plus the token.
    - If the token contains `foobar`, the content of the `Authorization` header would be: `Bearer foobar`.

FastAPI's `OAuth2PasswordBearer`

FastAPI provides several tools, at different levels of abstraction, to implement these security features.

In this example we are going to use **OAuth2**, with the **Password** flow, using a **Bearer** token. We do that using the `OAuth2PasswordBearer` class.

!!! info

    A "bearer" token is not the only option.

    But it's the best one for our use case.

    And it might be the best for most use cases, unless you are an OAuth2 expert and know exactly why there's another option that better suits your needs.

    In that case, **FastAPI** also provides you with the tools to build it.

When we create an instance of the `OAuth2PasswordBearer` class we pass in the `tokenUrl` parameter. This parameter contains the URL that the client (the frontend running in the user's browser) will use to send the `username` and `password` in order to get a token.

Python 3.9+

```python
from typing import Annotated

from fastapi import Depends, FastAPI
from fastapi.security import OAuth2PasswordBearer

app = FastAPI()

oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")


@app.get("/items/")
async def read_items(token: Annotated[str, Depends(oauth2_scheme)]):
    return {"token": token}
```

▸ 🤓 Other versions and variants

Python 3.8+

```python
from fastapi import Depends, FastAPI
from fastapi.security import OAuth2PasswordBearer
from typing_extensions import Annotated

app = FastAPI()

oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")


@app.get("/items/")
async def read_items(token: Annotated[str, Depends(oauth2_scheme)]):
    return {"token": token}
```

Python 3.8+ - non-Annotated

!!! tip

    Prefer to use the `Annotated` version if possible.

```python
from fastapi import Depends, FastAPI
from fastapi.security import OAuth2PasswordBearer

app = FastAPI()

oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")


@app.get("/items/")
async def read_items(token: str = Depends(oauth2_scheme)):
    return {"token": token}
```

Tip

Here `tokenUrl="token"` refers to a relative URL `token` that we haven't created yet. As it's a relative URL, it's equivalent to `./token`.

Because we are using a relative URL, if your API was located at `https://example.com/`, then it would refer to `https://example.com/token`. But if your API was located at `https://example.com/api/v1/`, then it would refer to `https://example.com/api/v1/token`.

Using a relative URL is important to make sure your application keeps working even in an advanced use case like Behind a Proxy.

This parameter doesn't create that endpoint / *path operation*, but declares that the URL `/token` will be the one that the client should use to get the token. That information is used in OpenAPI, and then in the interactive API documentation systems.

We will soon also create the actual path operation.

Info

If you are a very strict "Pythonista" you might dislike the style of the parameter name `tokenUrl` instead of `token_url`.

That's because it is using the same name as in the OpenAPI spec. So that if you need to investigate more about any of these security schemes you can just copy and paste it to find more information about it.

The `oauth2_scheme` variable is an instance of `OAuth2PasswordBearer`, but it is also a "callable".

It could be called as:

```
oauth2_scheme(some, parameters)
```

So, it can be used with `Depends`.

Use it

Now you can pass that `oauth2_scheme` in a dependency with `Depends`.

Python 3.9+

```
from typing import Annotated

from fastapi import Depends, FastAPI
from fastapi.security import OAuth2PasswordBearer

app = FastAPI()

oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")


@app.get("/items/")
async def read_items(token: Annotated[str, Depends(oauth2_scheme)]):
    return {"token": token}
```

▸ 🤓 Other versions and variants

Python 3.8+

```
from fastapi import Depends, FastAPI
from fastapi.security import OAuth2PasswordBearer
from typing_extensions import Annotated

app = FastAPI()

oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")


@app.get("/items/")
async def read_items(token: Annotated[str, Depends(oauth2_scheme)]):
    return {"token": token}
```

Python 3.8+ - non-Annotated

Tip

Prefer to use the `Annotated` version if possible.

```
from fastapi import Depends, FastAPI
from fastapi.security import OAuth2PasswordBearer

app = FastAPI()

oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")


@app.get("/items/")
async def read_items(token: str = Depends(oauth2_scheme)):
    return {"token": token}
```

This dependency will provide a `str` that is assigned to the parameter `token` of the *path operation function*.

FastAPI will know that it can use this dependency to define a "security scheme" in the OpenAPI schema (and the automatic API docs).

Technical Details

FastAPI will know that it can use the class `OAuth2PasswordBearer` (declared in a dependency) to define the security scheme in OpenAPI because it inherits from `fastapi.security.oauth2.OAuth2`, which in turn inherits from `fastapi.security.base.SecurityBase`.

All the security utilities that integrate with OpenAPI (and the automatic API docs) inherit from `SecurityBase`, that's how FastAPI can know how to integrate them in OpenAPI.

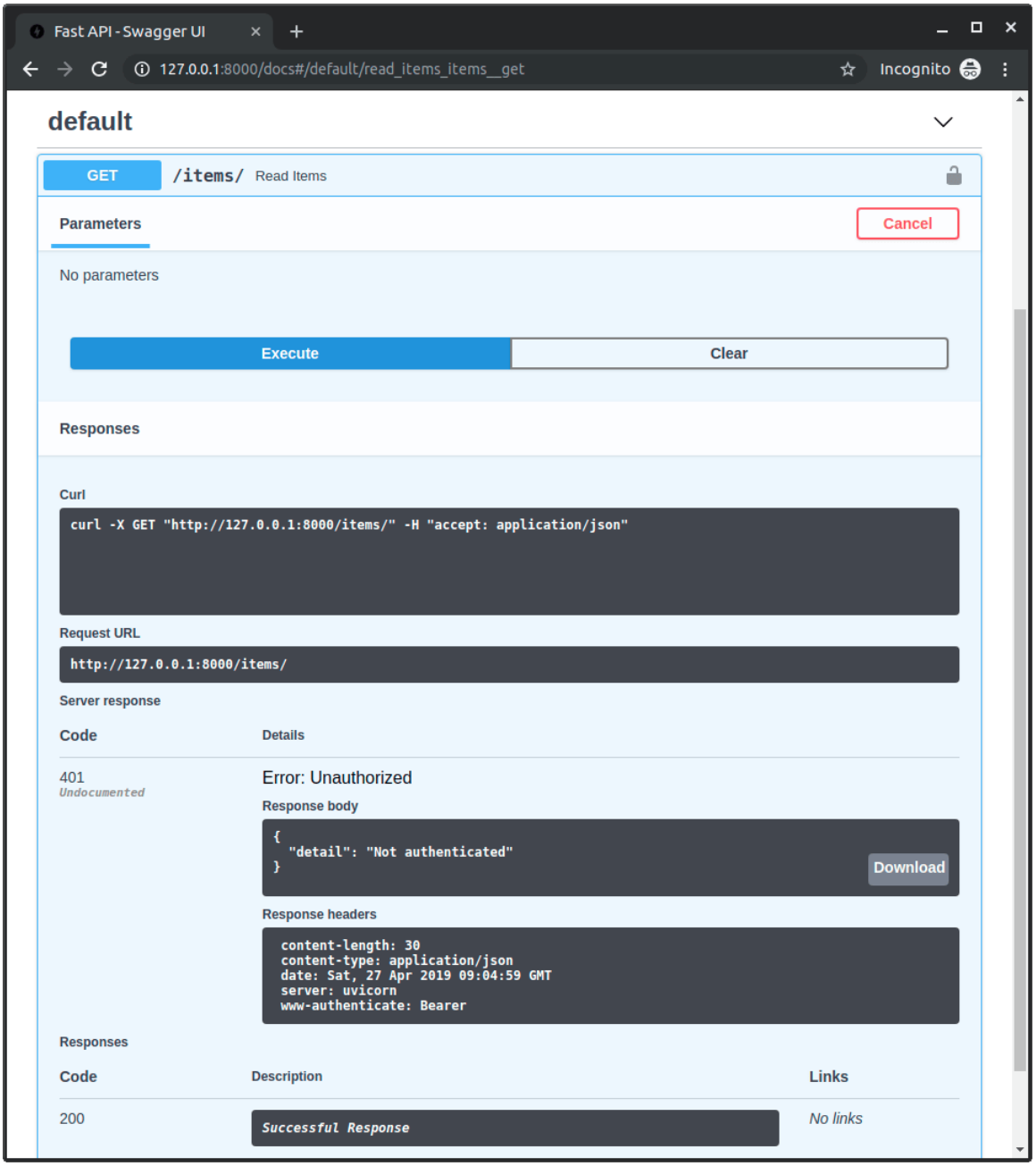
What it does

It will go and look in the request for that `Authorization` header, check if the value is `Bearer` plus some token, and will return the token as a `str`.

If it doesn't see an `Authorization` header, or the value doesn't have a `Bearer` token, it will respond with a 401 status code error (`UNAUTHORIZED`) directly.

You don't even have to check if the token exists to return an error. You can be sure that if your function is executed, it will have a `str` in that token.

You can try it already in the interactive docs:



We are not verifying the validity of the token yet, but that's a start already.

Recap

So, in just 3 or 4 extra lines, you already have some primitive form of security.