Security - First Steps[¶](https://fastapi.tiangolo.com/tutorial/security/first-steps/#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[¶](https://fastapi.tiangolo.com/tutorial/security/first-steps/#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[¶](https://fastapi.tiangolo.com/tutorial/security/first-steps/#create-mainpy)

Copy the example in a file main.py:



[Python 3.9+](https://fastapi.tiangolo.com/tutorial/security/first-steps/#__tabbed_1_1)

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}

**Info**

The [python-multipart](https://github.com/Kludex/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.

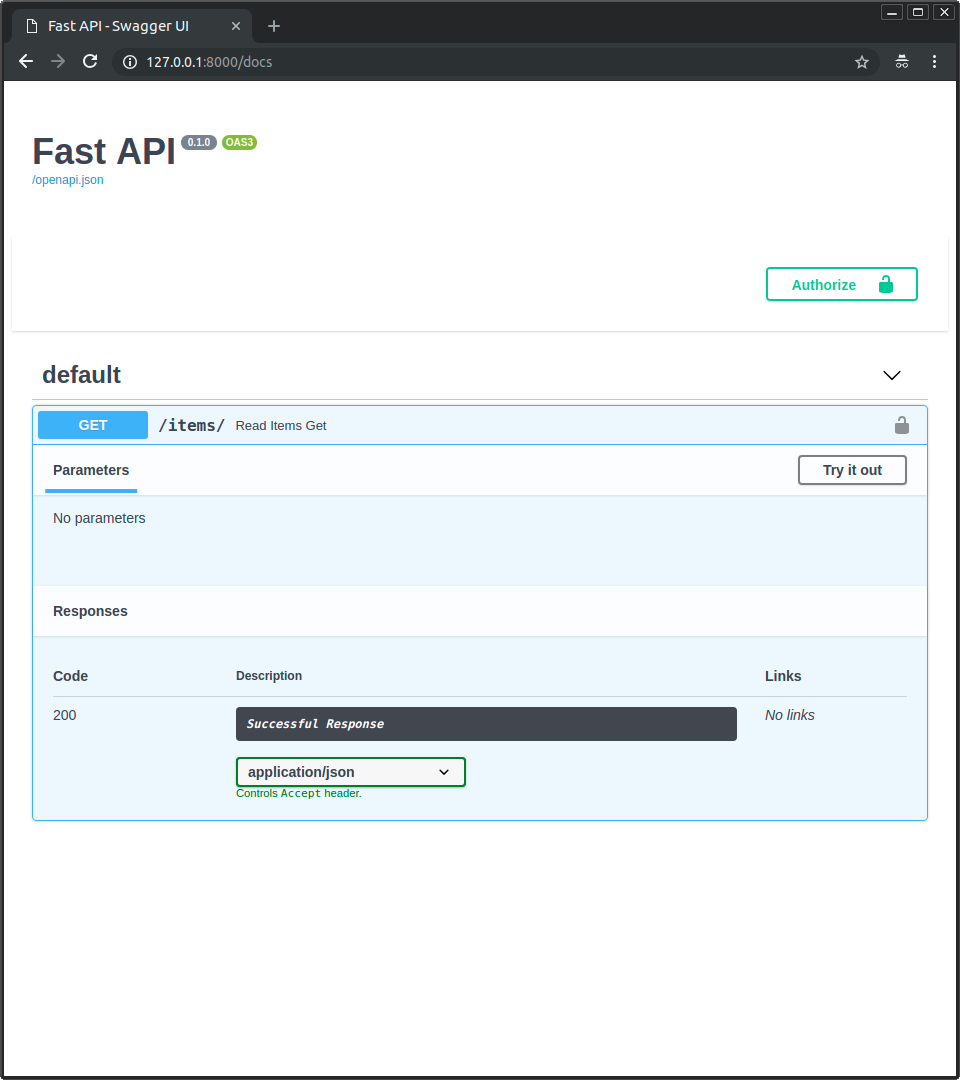
To install it manually, make sure you create a [virtual environment](https://fastapi.tiangolo.com/virtual-environments/), activate it, and then install it with:

$ 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  
INFO: Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)

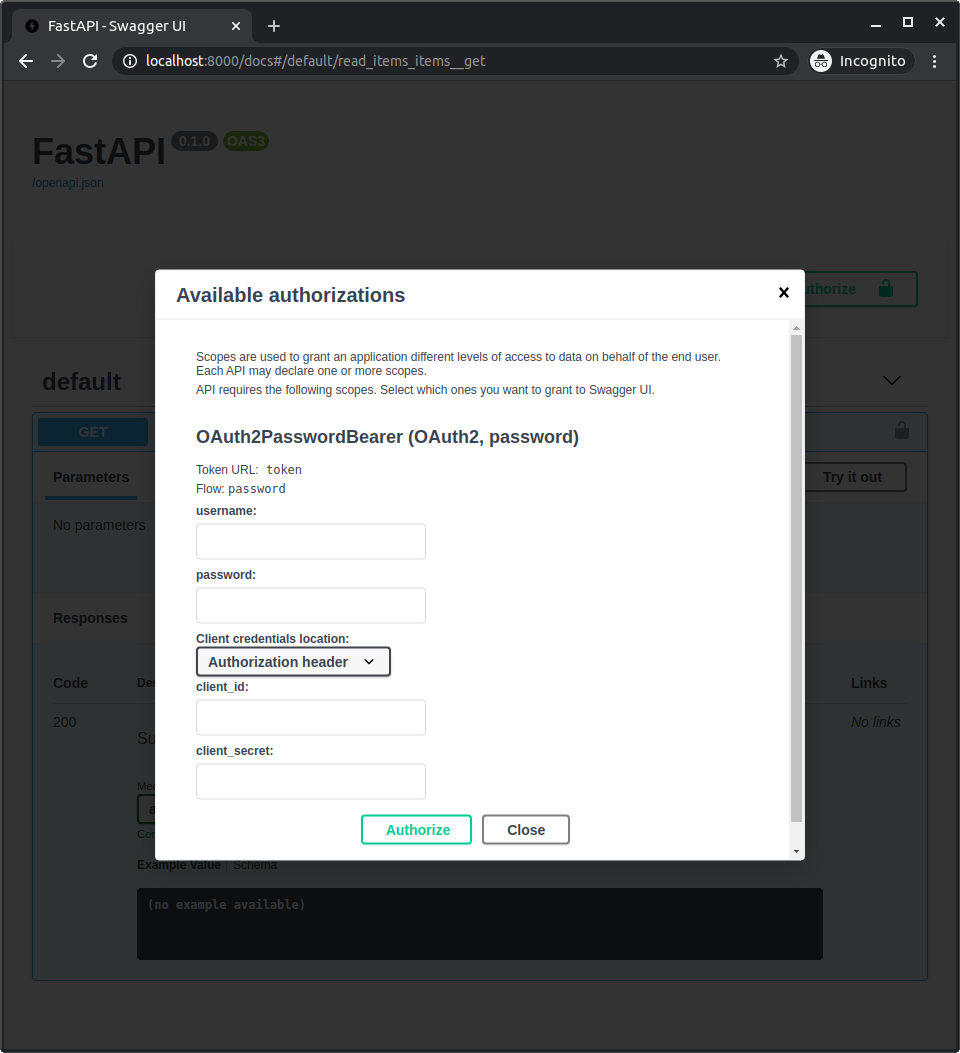


**Authorize button!**

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[¶](https://fastapi.tiangolo.com/tutorial/security/first-steps/#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 we use oauth2passwordbearer then system automatic bind token with form request header otherwise we have to bind using code or extra middleware system
  + 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.

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.

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}

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](https://fastapi.tiangolo.com/advanced/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.

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

What it does[¶](https://fastapi.tiangolo.com/tutorial/security/first-steps/#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.

# Get Current User

In the previous chapter the security system (which is based on the dependency injection system) was giving the path operation function a token as a str:



[Python 3.9+](https://fastapi.tiangolo.com/tutorial/security/get-current-user/#__tabbed_1_1)

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}

Let's make it give us the current user.

## Create a user model[¶](https://fastapi.tiangolo.com/tutorial/security/get-current-user/#create-a-user-model)

First, let's create a Pydantic user model.

The same way we use Pydantic to declare bodies, we can use it anywhere else:



[Python 3.10+](https://fastapi.tiangolo.com/tutorial/security/get-current-user/#__tabbed_3_1)

from typing import Annotated

from fastapi import Depends, FastAPI

from fastapi.security import OAuth2PasswordBearer

from pydantic import BaseModel

app = FastAPI()

oauth2\_scheme = OAuth2PasswordBearer(tokenUrl="token")

class User(BaseModel):

username: str

email: str | None = None

full\_name: str | None = None

disabled: bool | None = None

def fake\_decode\_token(token):

return User(

username=token + "fakedecoded", email="john@example.com", full\_name="John Doe"

)

async def get\_current\_user(token: Annotated[str, Depends(oauth2\_scheme)]):

user = fake\_decode\_token(token)

return user

@app.get("/users/me")

async def read\_users\_me(current\_user: Annotated[User, Depends(get\_current\_user)]):

return current\_user

## Create a get\_current\_user dependency

Let's create a dependency get\_current\_user.

Remember that dependencies can have sub-dependencies?

get\_current\_user will have a dependency with the same oauth2\_scheme we created before.

The same as we were doing before in the path operation directly, our new dependency get\_current\_user will receive a token as a str from the sub-dependency oauth2\_scheme:

from typing import Annotated

from fastapi import Depends, FastAPI

from fastapi.security import OAuth2PasswordBearer

from pydantic import BaseModel

app = FastAPI()

oauth2\_scheme = OAuth2PasswordBearer(tokenUrl="token")

class User(BaseModel):

username: str

email: str | None = None

full\_name: str | None = None

disabled: bool | None = None

def fake\_decode\_token(token):

return User(

username=token + "fakedecoded", email="john@example.com", full\_name="John Doe"

)

async def get\_current\_user(token: Annotated[str, Depends(oauth2\_scheme)]):

user = fake\_decode\_token(token)

return user

@app.get("/users/me")

async def read\_users\_me(current\_user: Annotated[User, Depends(get\_current\_user)]):

return current\_user

## Get the user

get\_current\_user will use a (fake) utility function we created, that takes a token as a str and returns our Pydantic User model:

from typing import Annotated

from fastapi import Depends, FastAPI

from fastapi.security import OAuth2PasswordBearer

from pydantic import BaseModel

app = FastAPI()

oauth2\_scheme = OAuth2PasswordBearer(tokenUrl="token")

class User(BaseModel):

username: str

email: str | None = None

full\_name: str | None = None

disabled: bool | None = None

def fake\_decode\_token(token):

return User(

username=token + "fakedecoded", email="john@example.com", full\_name="John Doe"

)

async def get\_current\_user(token: Annotated[str, Depends(oauth2\_scheme)]):

user = fake\_decode\_token(token)

return user

@app.get("/users/me")

async def read\_users\_me(current\_user: Annotated[User, Depends(get\_current\_user)]):

return current\_user

## Inject the current user[¶](https://fastapi.tiangolo.com/tutorial/security/get-current-user/#inject-the-current-user)

So now we can use the same Depends with our get\_current\_user in the path operation:



[Python 3.10+](https://fastapi.tiangolo.com/tutorial/security/get-current-user/#__tabbed_9_1)

from typing import Annotated

from fastapi import Depends, FastAPI

from fastapi.security import OAuth2PasswordBearer

from pydantic import BaseModel

app = FastAPI()

oauth2\_scheme = OAuth2PasswordBearer(tokenUrl="token")

class User(BaseModel):

username: str

email: str | None = None

full\_name: str | None = None

disabled: bool | None = None

def fake\_decode\_token(token):

return User(

username=token + "fakedecoded", email="john@example.com", full\_name="John Doe"

)

async def get\_current\_user(token: Annotated[str, Depends(oauth2\_scheme)]):

user = fake\_decode\_token(token)

return user

@app.get("/users/me")

async def read\_users\_me(current\_user: Annotated[User, Depends(get\_current\_user)]):

return current\_user

Notice that we declare the type of current\_user as the Pydantic model User.

This will help us inside of the function with all the completion and type checks.

**Tip**

You might remember that request bodies are also declared with Pydantic models.

Here **FastAPI** won't get confused because you are using Depends.

**Check**

The way this dependency system is designed allows us to have different dependencies (different "dependables") that all return a User model.

We are not restricted to having only one dependency that can return that type of data.

## Other models

You can now get the current user directly in the path operation functions and deal with the security mechanisms at the **Dependency Injection** level, using Depends.

And you can use any model or data for the security requirements (in this case, a Pydantic model User).

But you are not restricted to using some specific data model, class or type.

Do you want to have an id and email and not have any username in your model? Sure. You can use these same tools.

Do you want to just have a str? Or just a dict? Or a database class model instance directly? It all works the same way.

You actually don't have users that log in to your application but robots, bots, or other systems, that have just an access token? Again, it all works the same.

Just use any kind of model, any kind of class, any kind of database that you need for your application. **FastAPI** has you covered with the dependency injection system.

## Code size[¶](https://fastapi.tiangolo.com/tutorial/security/get-current-user/#code-size)

This example might seem verbose. Keep in mind that we are mixing security, data models, utility functions and path operations in the same file.

But here's the key point.

The security and dependency injection stuff is written once.

And you can make it as complex as you want. And still, have it written only once, in a single place. With all the flexibility.

But you can have thousands of endpoints (path operations) using the same security system.

And all of them (or any portion of them that you want) can take advantage of re-using these dependencies or any other dependencies you create.

And all these thousands of path operations can be as small as 3 lines:



[Python 3.10+](https://fastapi.tiangolo.com/tutorial/security/get-current-user/#__tabbed_11_1)

from typing import Annotated

from fastapi import Depends, FastAPI

from fastapi.security import OAuth2PasswordBearer

from pydantic import BaseModel

app = FastAPI()

oauth2\_scheme = OAuth2PasswordBearer(tokenUrl="token")

class User(BaseModel):

username: str

email: str | None = None

full\_name: str | None = None

disabled: bool | None = None

def fake\_decode\_token(token):

return User(

username=token + "fakedecoded", email="john@example.com", full\_name="John Doe"

)

async def get\_current\_user(token: Annotated[str, Depends(oauth2\_scheme)]):

user = fake\_decode\_token(token)

return user

@app.get("/users/me")

async def read\_users\_me(current\_user: Annotated[User, Depends(get\_current\_user)]):

return current\_user

You can now get the current user directly in your path operation function.

We are already halfway there.

We just need to add a path operation for the user/client to actually send the username and password.

**Simple OAuth2 with Password and Bearer**

Now let's build from the previous chapter and add the missing parts to have a complete security flow. We are going to use **FastAPI** security utilities to get the username and password.

OAuth2 specifies that when using the "password flow" (that we are using) the client/user must send a username and password fields as form data.

And the spec says that the fields have to be named like that. So user-name or email wouldn't work.

But don't worry, you can show it as you wish to your final users in the frontend.

And your database models can use any other names you want.

But for the login *path operation*, we need to use these names to be compatible with the spec (and be able to, for example, use the integrated API documentation system).

The spec also states that the username and password must be sent as form data (so, no JSON here).

The spec also says that the client can send another form field "scope".

The form field name is scope (in singular), but it is actually a long string with "scopes" separated by spaces.

Each "scope" is just a string (without spaces).

They are normally used to declare specific security permissions, for example:

* users:read or users:write are common examples.
* instagram\_basic is used by Facebook / Instagram.
* https://www.googleapis.com/auth/drive is used by Google.

**Info**

In OAuth2 a "scope" is just a string that declares a specific permission required.

It doesn't matter if it has other characters like : or if it is a URL.

Those details are implementation specific.

For OAuth2 they are just strings.

Code to get the username and password[¶](https://fastapi.tiangolo.com/tutorial/security/simple-oauth2/#code-to-get-the-username-and-password)

Now let's use the utilities provided by **FastAPI** to handle this.

OAuth2PasswordRequestForm[¶](https://fastapi.tiangolo.com/tutorial/security/simple-oauth2/#oauth2passwordrequestform)

First, import OAuth2PasswordRequestForm, and use it as a dependency with Depends in the *path operation* for /token:

from typing import Annotated

from fastapi import Depends, FastAPI, HTTPException, status

from fastapi.security import OAuth2PasswordBearer, OAuth2PasswordRequestForm

from pydantic import BaseModel

fake\_users\_db = {

"johndoe": {

"username": "johndoe",

"full\_name": "John Doe",

"email": "johndoe@example.com",

"hashed\_password": "fakehashedsecret",

"disabled": False,

},

"alice": {

"username": "alice",

"full\_name": "Alice Wonderson",

"email": "alice@example.com",

"hashed\_password": "fakehashedsecret2",

"disabled": True,

},

}

app = FastAPI()

def fake\_hash\_password(password: str):

return "fakehashed" + password

oauth2\_scheme = OAuth2PasswordBearer(tokenUrl="token")

class User(BaseModel):

username: str

email: str | None = None

full\_name: str | None = None

disabled: bool | None = None

class UserInDB(User):

hashed\_password: str

def get\_user(db, username: str):

if username in db:

user\_dict = db[username]

return UserInDB(\*\*user\_dict)

def fake\_decode\_token(token):

# This doesn't provide any security at all

# Check the next version

user = get\_user(fake\_users\_db, token)

return user

async def get\_current\_user(token: Annotated[str, Depends(oauth2\_scheme)]):

user = fake\_decode\_token(token)

if not user:

raise HTTPException(

status\_code=status.HTTP\_401\_UNAUTHORIZED,

detail="Invalid authentication credentials",

headers={"WWW-Authenticate": "Bearer"},

)

return user

async def get\_current\_active\_user(

current\_user: Annotated[User, Depends(get\_current\_user)],

):

if current\_user.disabled:

raise HTTPException(status\_code=400, detail="Inactive user")

return current\_user

@app.post("/token")

async def login(form\_data: Annotated[OAuth2PasswordRequestForm, Depends()]):

user\_dict = fake\_users\_db.get(form\_data.username)

if not user\_dict:

raise HTTPException(status\_code=400, detail="Incorrect username or password")

user = UserInDB(\*\*user\_dict)

hashed\_password = fake\_hash\_password(form\_data.password)

if not hashed\_password == user.hashed\_password:

raise HTTPException(status\_code=400, detail="Incorrect username or password")

return {"access\_token": user.username, "token\_type": "bearer"}

@app.get("/users/me")

async def read\_users\_me(

current\_user: Annotated[User, Depends(get\_current\_active\_user)],

):

return current\_user

OAuth2PasswordRequestForm is a class dependency that declares a form body with:

* The username.
* The password.
* An optional scope field as a big string, composed of strings separated by spaces.
* An optional grant\_type.

**Tip**

The OAuth2 spec actually *requires* a field grant\_type with a fixed value of password, but OAuth2PasswordRequestForm doesn't enforce it.If you need to enforce it, use OAuth2PasswordRequestFormStrict instead of OAuth2PasswordRequestForm.

* An optional client\_id (we don't need it for our example).
* An optional client\_secret (we don't need it for our example).

**Info**

The OAuth2PasswordRequestForm is not a special class for **FastAPI** as is OAuth2PasswordBearer.

OAuth2PasswordBearer makes **FastAPI** know that it is a security scheme. So it is added that way to OpenAPI.

But OAuth2PasswordRequestForm is just a class dependency that you could have written yourself, or you could have declared Form parameters directly.

But as it's a common use case, it is provided by **FastAPI** directly, just to make it easier.

Use the form data[¶](https://fastapi.tiangolo.com/tutorial/security/simple-oauth2/#use-the-form-data)

**Tip**

The instance of the dependency class OAuth2PasswordRequestForm won't have an attribute scope with the long string separated by spaces, instead, it will have a scopes attribute with the actual list of strings for each scope sent.

We are not using scopes in this example, but the functionality is there if you need it.

Now, get the user data from the (fake) database, using the username from the form field.

If there is no such user, we return an error saying "Incorrect username or password".

For the error, we use the exception HTTPException:

from typing import Annotated

from fastapi import Depends, FastAPI, HTTPException, status

from fastapi.security import OAuth2PasswordBearer, OAuth2PasswordRequestForm

from pydantic import BaseModel

fake\_users\_db = {

"johndoe": {

"username": "johndoe",

"full\_name": "John Doe",

"email": "johndoe@example.com",

"hashed\_password": "fakehashedsecret",

"disabled": False,

},

"alice": {

"username": "alice",

"full\_name": "Alice Wonderson",

"email": "alice@example.com",

"hashed\_password": "fakehashedsecret2",

"disabled": True,

},

}

app = FastAPI()

def fake\_hash\_password(password: str):

return "fakehashed" + password

oauth2\_scheme = OAuth2PasswordBearer(tokenUrl="token")

class User(BaseModel):

username: str

email: str | None = None

full\_name: str | None = None

disabled: bool | None = None

class UserInDB(User):

hashed\_password: str

def get\_user(db, username: str):

if username in db:

user\_dict = db[username]

return UserInDB(\*\*user\_dict)

def fake\_decode\_token(token):

# This doesn't provide any security at all

# Check the next version

user = get\_user(fake\_users\_db, token)

return user

async def get\_current\_user(token: Annotated[str, Depends(oauth2\_scheme)]):

user = fake\_decode\_token(token)

if not user:

raise HTTPException(

status\_code=status.HTTP\_401\_UNAUTHORIZED,

detail="Invalid authentication credentials",

headers={"WWW-Authenticate": "Bearer"},

)

return user

async def get\_current\_active\_user(

current\_user: Annotated[User, Depends(get\_current\_user)],

):

if current\_user.disabled:

raise HTTPException(status\_code=400, detail="Inactive user")

return current\_user

@app.post("/token")

async def login(form\_data: Annotated[OAuth2PasswordRequestForm, Depends()]):

user\_dict = fake\_users\_db.get(form\_data.username)

if not user\_dict:

raise HTTPException(status\_code=400, detail="Incorrect username or password")

user = UserInDB(\*\*user\_dict)

hashed\_password = fake\_hash\_password(form\_data.password)

if not hashed\_password == user.hashed\_password:

raise HTTPException(status\_code=400, detail="Incorrect username or password")

return {"access\_token": user.username, "token\_type": "bearer"}

@app.get("/users/me")

async def read\_users\_me(

current\_user: Annotated[User, Depends(get\_current\_active\_user)],

):

return current\_user

Check the password[¶](https://fastapi.tiangolo.com/tutorial/security/simple-oauth2/#check-the-password)

At this point we have the user data from our database, but we haven't checked the password.

Let's put that data in the Pydantic UserInDB model first.

You should never save plaintext passwords, so, we'll use the (fake) password hashing system.

If the passwords don't match, we return the same error.

**Password hashing**[**¶**](https://fastapi.tiangolo.com/tutorial/security/simple-oauth2/#password-hashing)

"Hashing" means: converting some content (a password in this case) into a sequence of bytes (just a string) that looks like gibberish.

Whenever you pass exactly the same content (exactly the same password) you get exactly the same gibberish.

But you cannot convert from the gibberish back to the password.

**Why use password hashing**[**¶**](https://fastapi.tiangolo.com/tutorial/security/simple-oauth2/#why-use-password-hashing)

If your database is stolen, the thief won't have your users' plaintext passwords, only the hashes.

So, the thief won't be able to try to use those same passwords in another system (as many users use the same password everywhere, this would be dangerous).



[Python 3.10+](https://fastapi.tiangolo.com/tutorial/security/simple-oauth2/#__tabbed_5_1)

from typing import Annotated

from fastapi import Depends, FastAPI, HTTPException, status

from fastapi.security import OAuth2PasswordBearer, OAuth2PasswordRequestForm

from pydantic import BaseModel

fake\_users\_db = {

"johndoe": {

"username": "johndoe",

"full\_name": "John Doe",

"email": "johndoe@example.com",

"hashed\_password": "fakehashedsecret",

"disabled": False,

},

"alice": {

"username": "alice",

"full\_name": "Alice Wonderson",

"email": "alice@example.com",

"hashed\_password": "fakehashedsecret2",

"disabled": True,

},

}

app = FastAPI()

def fake\_hash\_password(password: str):

return "fakehashed" + password

oauth2\_scheme = OAuth2PasswordBearer(tokenUrl="token")

class User(BaseModel):

username: str

email: str | None = None

full\_name: str | None = None

disabled: bool | None = None

class UserInDB(User):

hashed\_password: str

def get\_user(db, username: str):

if username in db:

user\_dict = db[username]

return UserInDB(\*\*user\_dict)

def fake\_decode\_token(token):

# This doesn't provide any security at all

# Check the next version

user = get\_user(fake\_users\_db, token)

return user

async def get\_current\_user(token: Annotated[str, Depends(oauth2\_scheme)]):

user = fake\_decode\_token(token)

if not user:

raise HTTPException(

status\_code=status.HTTP\_401\_UNAUTHORIZED,

detail="Invalid authentication credentials",

headers={"WWW-Authenticate": "Bearer"},

)

return user

async def get\_current\_active\_user(

current\_user: Annotated[User, Depends(get\_current\_user)],

):

if current\_user.disabled:

raise HTTPException(status\_code=400, detail="Inactive user")

return current\_user

@app.post("/token")

async def login(form\_data: Annotated[OAuth2PasswordRequestForm, Depends()]):

user\_dict = fake\_users\_db.get(form\_data.username)

if not user\_dict:

raise HTTPException(status\_code=400, detail="Incorrect username or password")

user = UserInDB(\*\*user\_dict)

hashed\_password = fake\_hash\_password(form\_data.password)

if not hashed\_password == user.hashed\_password:

raise HTTPException(status\_code=400, detail="Incorrect username or password")

return {"access\_token": user.username, "token\_type": "bearer"}

@app.get("/users/me")

async def read\_users\_me(

current\_user: Annotated[User, Depends(get\_current\_active\_user)],

):

return current\_user

🤓 Other versions and variants



**About \*\*user\_dict[¶](https://fastapi.tiangolo.com/tutorial/security/simple-oauth2/" \l "about-user_dict" \o "Permanent link)**

UserInDB(\*\*user\_dict) means:

*Pass the keys and values of the user\_dict directly as key-value arguments, equivalent to:*

UserInDB(

username = user\_dict["username"],

email = user\_dict["email"],

full\_name = user\_dict["full\_name"],

disabled = user\_dict["disabled"],

hashed\_password = user\_dict["hashed\_password"],

)

**Info**

For a more complete explanation of \*\*user\_dict check back in [the documentation for **Extra Models**](https://fastapi.tiangolo.com/tutorial/extra-models/#about-user_indict).

Pydantic models have a .dict() method that returns a dict with the model's data. user\_dict = user\_dict.dict()

Return the token

The response of the token endpoint must be a JSON object.

It should have a token\_type. In our case, as we are using "Bearer" tokens, the token type should be "bearer".

And it should have an access\_token, with a string containing our access token.

For this simple example, we are going to just be completely insecure and return the same username as the token.

**Tip**

In the next chapter, you will see a real secure implementation, with password hashing and JWT tokens.

But for now, let's focus on the specific details we need.



[Python 3.10+](https://fastapi.tiangolo.com/tutorial/security/simple-oauth2/#__tabbed_7_1)

from typing import Annotated

from fastapi import Depends, FastAPI, HTTPException, status

from fastapi.security import OAuth2PasswordBearer, OAuth2PasswordRequestForm

from pydantic import BaseModel

fake\_users\_db = {

"johndoe": {

"username": "johndoe",

"full\_name": "John Doe",

"email": "johndoe@example.com",

"hashed\_password": "fakehashedsecret",

"disabled": False,

},

"alice": {

"username": "alice",

"full\_name": "Alice Wonderson",

"email": "alice@example.com",

"hashed\_password": "fakehashedsecret2",

"disabled": True,

},

}

app = FastAPI()

def fake\_hash\_password(password: str):

return "fakehashed" + password

oauth2\_scheme = OAuth2PasswordBearer(tokenUrl="token")

class User(BaseModel):

username: str

email: str | None = None

full\_name: str | None = None

disabled: bool | None = None

class UserInDB(User):

hashed\_password: str

def get\_user(db, username: str):

if username in db:

user\_dict = db[username]

return UserInDB(\*\*user\_dict)

def fake\_decode\_token(token):

# This doesn't provide any security at all

# Check the next version

user = get\_user(fake\_users\_db, token)

return user

async def get\_current\_user(token: Annotated[str, Depends(oauth2\_scheme)]):

user = fake\_decode\_token(token)

if not user:

raise HTTPException(

status\_code=status.HTTP\_401\_UNAUTHORIZED,

detail="Invalid authentication credentials",

headers={"WWW-Authenticate": "Bearer"},

)

return user

async def get\_current\_active\_user(

current\_user: Annotated[User, Depends(get\_current\_user)],

):

if current\_user.disabled:

raise HTTPException(status\_code=400, detail="Inactive user")

return current\_user

@app.post("/token")

async def login(form\_data: Annotated[OAuth2PasswordRequestForm, Depends()]):

user\_dict = fake\_users\_db.get(form\_data.username)

if not user\_dict:

raise HTTPException(status\_code=400, detail="Incorrect username or password")

user = UserInDB(\*\*user\_dict)

hashed\_password = fake\_hash\_password(form\_data.password)

if not hashed\_password == user.hashed\_password:

raise HTTPException(status\_code=400, detail="Incorrect username or password")

return {"access\_token": user.username, "token\_type": "bearer"}

@app.get("/users/me")

async def read\_users\_me(

current\_user: Annotated[User, Depends(get\_current\_active\_user)],

):

return current\_user

By the spec, you should return a JSON with an access\_token and a token\_type, the same as in this example.

This is something that you have to do yourself in your code, and make sure you use those JSON keys.

It's almost the only thing that you have to remember to do correctly yourself, to be compliant with the specifications.

For the rest, **FastAPI** handles it for you.

Update the dependencies[¶](https://fastapi.tiangolo.com/tutorial/security/simple-oauth2/#update-the-dependencies)

Now we are going to update our dependencies.

We want to get the current\_user *only* if this user is active.

So, we create an additional dependency get\_current\_active\_user that in turn uses get\_current\_user as a dependency.

Both of these dependencies will just return an HTTP error if the user doesn't exist, or if is inactive.

So, in our endpoint, we will only get a user if the user exists, was correctly authenticated, and is active:



[Python 3.10+](https://fastapi.tiangolo.com/tutorial/security/simple-oauth2/#__tabbed_9_1)

from typing import Annotated

from fastapi import Depends, FastAPI, HTTPException, status

from fastapi.security import OAuth2PasswordBearer, OAuth2PasswordRequestForm

from pydantic import BaseModel

fake\_users\_db = {

"johndoe": {

"username": "johndoe",

"full\_name": "John Doe",

"email": "johndoe@example.com",

"hashed\_password": "fakehashedsecret",

"disabled": False,

},

"alice": {

"username": "alice",

"full\_name": "Alice Wonderson",

"email": "alice@example.com",

"hashed\_password": "fakehashedsecret2",

"disabled": True,

},

}

app = FastAPI()

def fake\_hash\_password(password: str):

return "fakehashed" + password

oauth2\_scheme = OAuth2PasswordBearer(tokenUrl="token")

class User(BaseModel):

username: str

email: str | None = None

full\_name: str | None = None

disabled: bool | None = None

class UserInDB(User):

hashed\_password: str

def get\_user(db, username: str):

if username in db:

user\_dict = db[username]

return UserInDB(\*\*user\_dict)

def fake\_decode\_token(token):

# This doesn't provide any security at all

# Check the next version

user = get\_user(fake\_users\_db, token)

return user

async def get\_current\_user(token: Annotated[str, Depends(oauth2\_scheme)]):

user = fake\_decode\_token(token)

if not user:

raise HTTPException(

status\_code=status.HTTP\_401\_UNAUTHORIZED,

detail="Invalid authentication credentials",

headers={"WWW-Authenticate": "Bearer"},

)

return user

async def get\_current\_active\_user(

current\_user: Annotated[User, Depends(get\_current\_user)],

):

if current\_user.disabled:

raise HTTPException(status\_code=400, detail="Inactive user")

return current\_user

@app.post("/token")

async def login(form\_data: Annotated[OAuth2PasswordRequestForm, Depends()]):

user\_dict = fake\_users\_db.get(form\_data.username)

if not user\_dict:

raise HTTPException(status\_code=400, detail="Incorrect username or password")

user = UserInDB(\*\*user\_dict)

hashed\_password = fake\_hash\_password(form\_data.password)

if not hashed\_password == user.hashed\_password:

raise HTTPException(status\_code=400, detail="Incorrect username or password")

return {"access\_token": user.username, "token\_type": "bearer"}

@app.get("/users/me")

async def read\_users\_me(

current\_user: Annotated[User, Depends(get\_current\_active\_user)],

):

return current\_user

🤓 Other versions and variants



**Info**

The additional header WWW-Authenticate with value Bearer we are returning here is also part of the spec.

Any HTTP (error) status code 401 "UNAUTHORIZED" is supposed to also return a WWW-Authenticate header.

In the case of bearer tokens (our case), the value of that header should be Bearer.

You can actually skip that extra header and it would still work.

But it's provided here to be compliant with the specifications.

Also, there might be tools that expect and use it (now or in the future) and that might be useful for you or your users, now or in the future.

That's the benefit of standards...

See it in action[¶](https://fastapi.tiangolo.com/tutorial/security/simple-oauth2/#see-it-in-action)

Open the interactive docs: <http://127.0.0.1:8000/docs>.

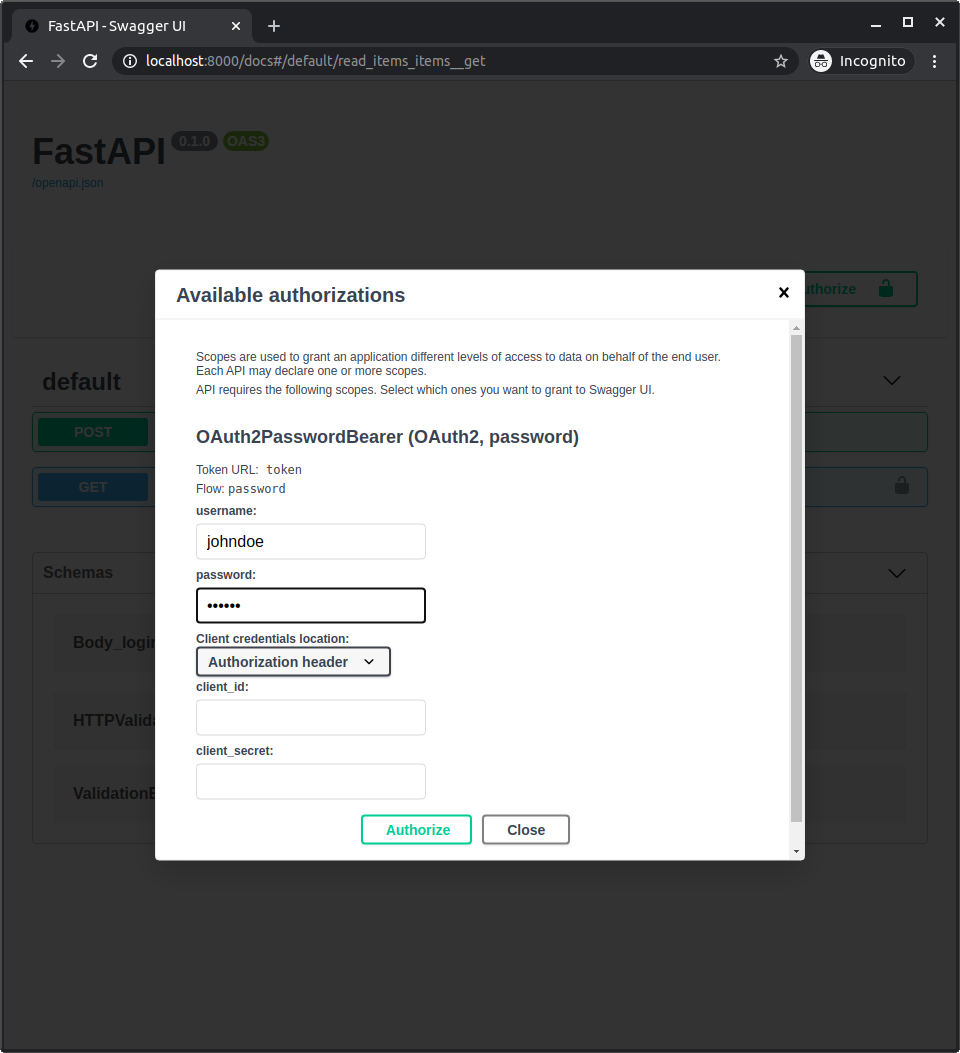
Authenticate[¶](https://fastapi.tiangolo.com/tutorial/security/simple-oauth2/#authenticate)

Click the "Authorize" button.

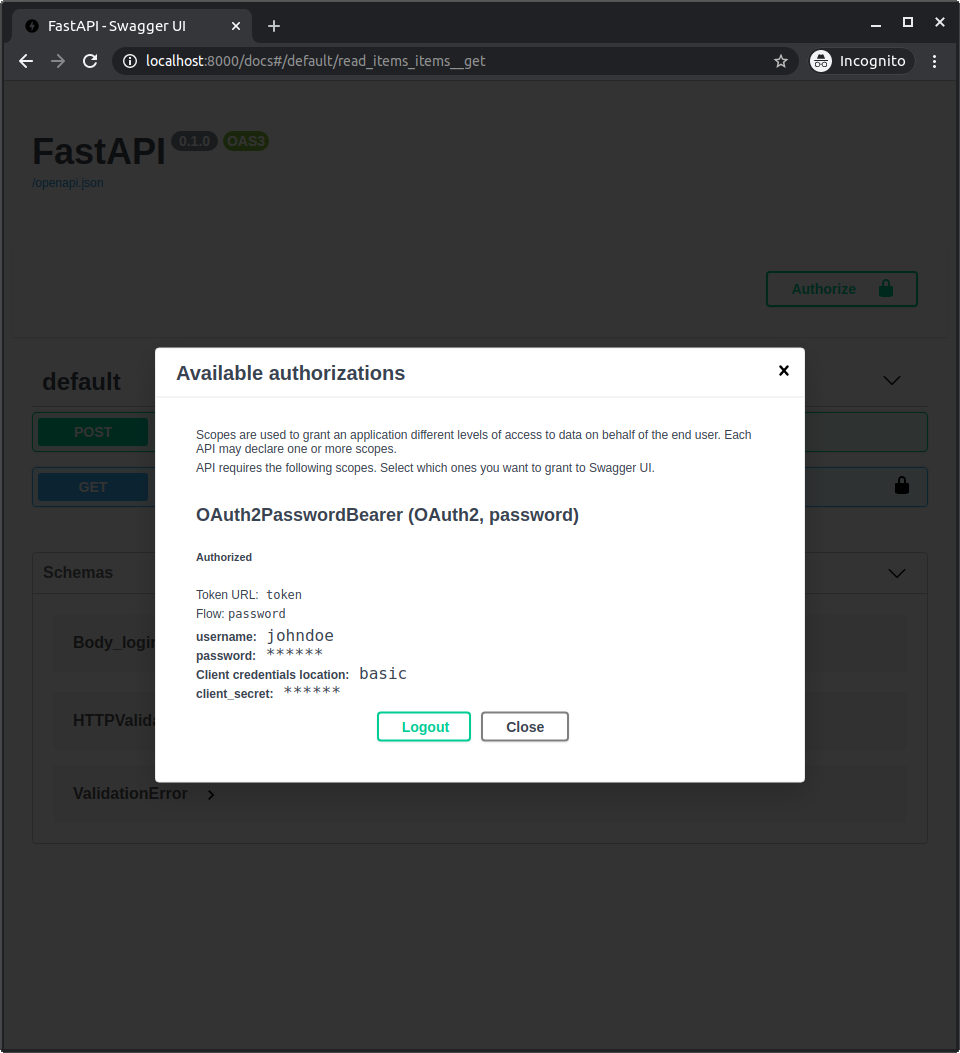
Use the credentials:

User: johndoe

Password: secret



After authenticating in the system, you will see it like:



Get your own user data[¶](https://fastapi.tiangolo.com/tutorial/security/simple-oauth2/#get-your-own-user-data)

Now use the operation GET with the path /users/me.

You will get your user's data, like:

{

"username": "johndoe",

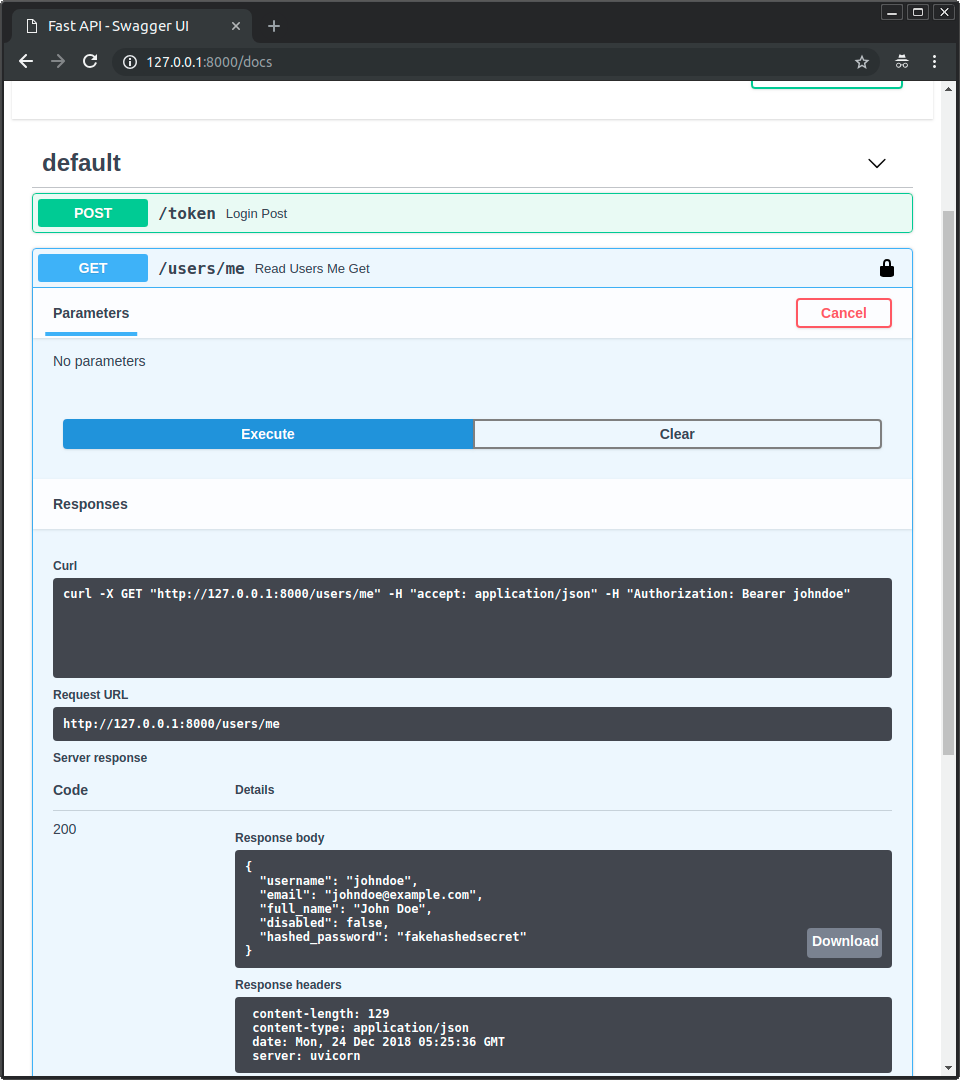
"email": "johndoe@example.com",

"full\_name": "John Doe",

"disabled": false,

"hashed\_password": "fakehashedsecret"

}



If you click the lock icon and logout, and then try the same operation again, you will get an HTTP 401 error of:

{

"detail": "Not authenticated"

}

Inactive user[¶](https://fastapi.tiangolo.com/tutorial/security/simple-oauth2/#inactive-user)

Now try with an inactive user, authenticate with:

User: alice

Password: secret2

And try to use the operation GET with the path /users/me.

You will get an "Inactive user" error, like:

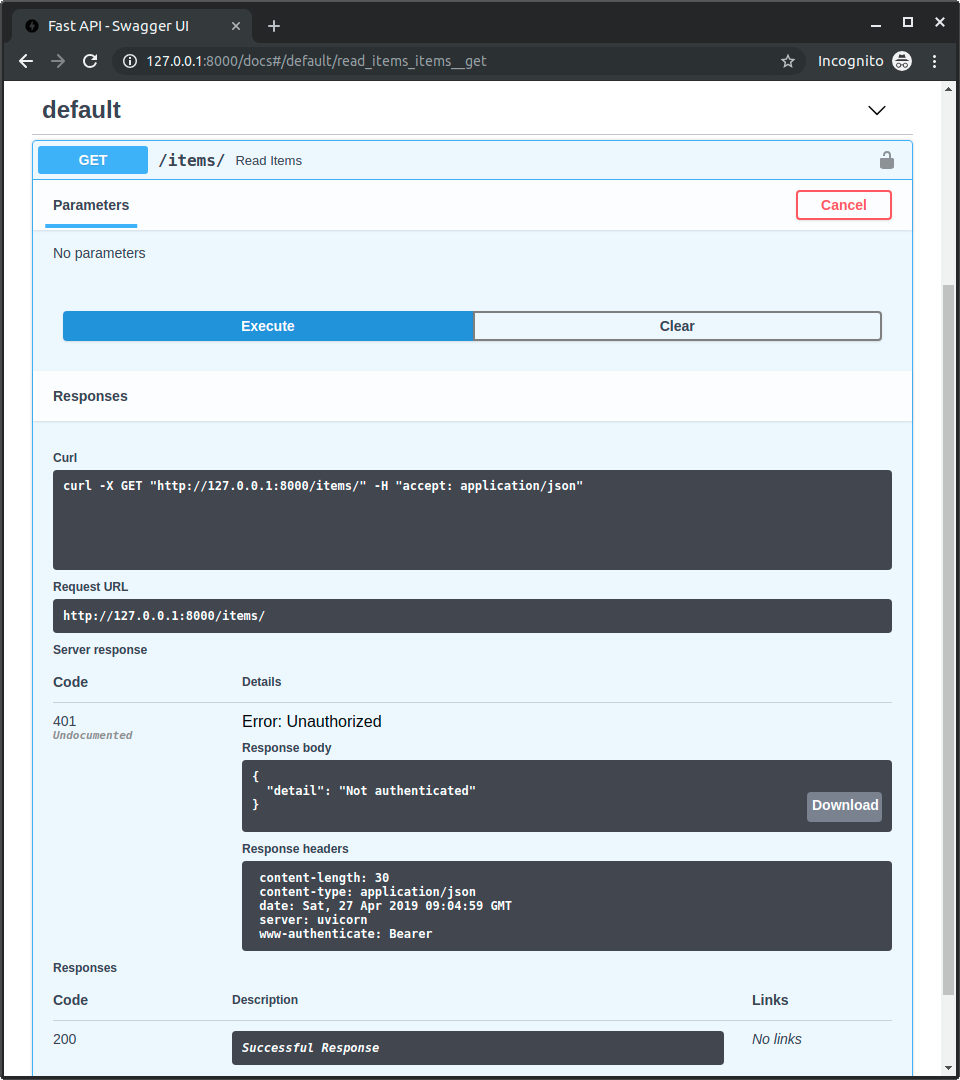
{

"detail": "Inactive user"

}

Recap[¶](https://fastapi.tiangolo.com/tutorial/security/simple-oauth2/#recap)

You now have the tools to implement a complete security system based on username and password for your API.



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