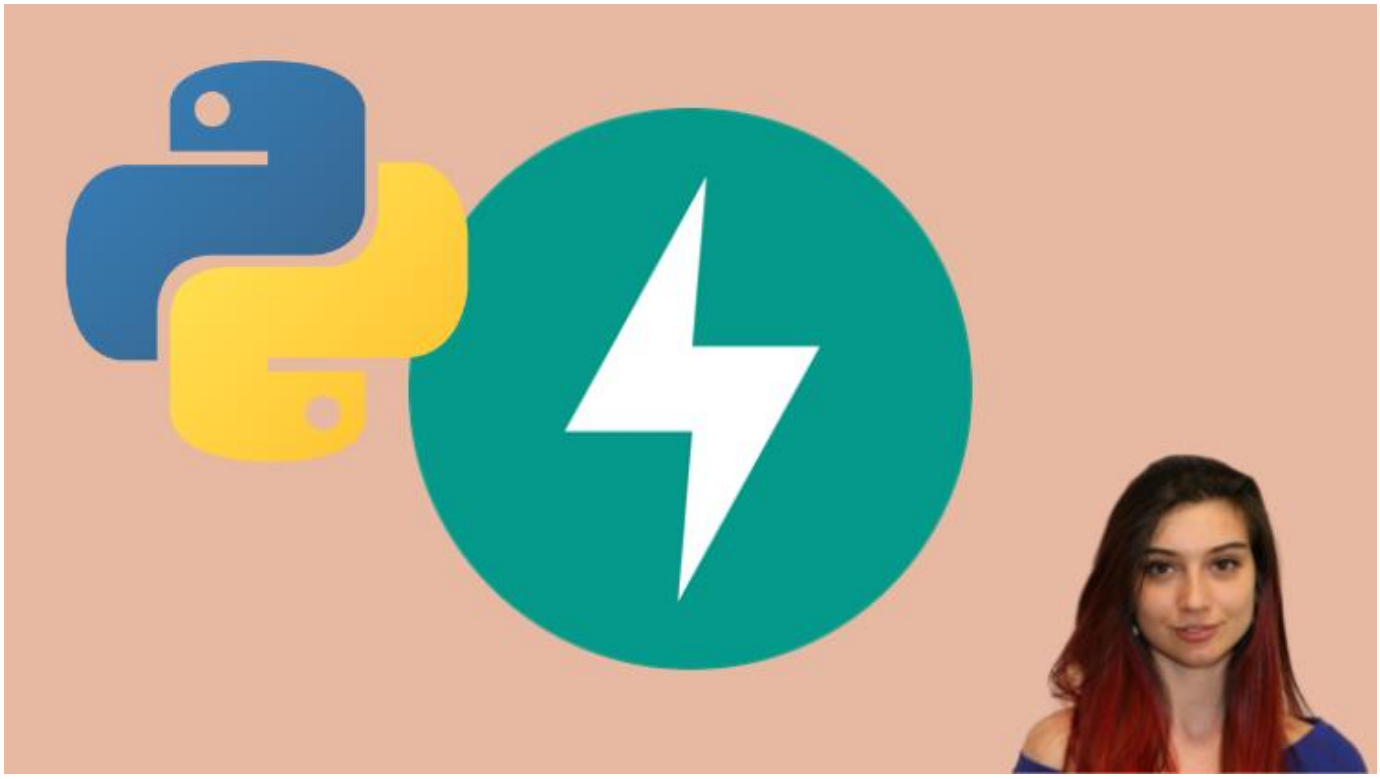
FastAPI REST – Part 4



CODE WITH FINESSE®

Ready to go?

Authentication and authorization

TABLE OF CONTENTS

1. JW

1.1 Definition

JSON Web Tokens (or JWTs) provide a means of transmitting information from the client to the server in a stateless, secure way.

On the server, JWTs are generated by signing user information via a secret key, which are then securely stored on the client. This form of auth works well with modern, single page applications. For more on this, along with the pros and cons of using JWTs vs. session and cookie-based auth, please review the following article.
The auth workflow works as follows:

- Client provides email and password, which is sent to the server
- Server then verifies that email and password are correct and responds with an auth token
- Client stores the token and sends it along with all subsequent requests to the API
- Server decodes the token and validates it on each request
- This cycle repeats until the token expires or is revoked. In the latter case, the server issues a new token.

The tokens themselves are divided into three parts:

- Header
- Payload
- Signature

you can read more about each part from here.

1.2 Encode token

To work with JSON Web Tokens in our app:

```
pip install pyjwt
```

We will need a function – for encoding the jwt.
It will help us to create a token. Here is the part where we have to set the token's expiration time and the so-called 'sub' (stands for subject). In most cases, the sub is the id of the user. Depending on the application security's requirements, the token can expire in 5 minutes or five days. Of course, you can set it never to expire, but this is not a good idea from a security perspective.
You can decide where to define this function. But for now, because we are working in single file, you should place it in *main.py*

The JWT_SECRET is a value from .env file. You can set it to whatever you want, but please do not change it, once you have set it.

```
def create_access_token(user):
    try:
        payload = {"sub": user["id"], "exp":  datetime.utcnow() +
timedelta(minutes=120)}
        return jwt.encode(payload, config('JWT_SECRET'),
algorithm='HS256')
    except Exception as e:
        raise e
```

In the payload, we define the **exp** key and the **sub** key. We use the user's id for the subject, and we set the exp date for two days. You can also define this in the .env file, so developers do not need to change it and commit it to production by mistake.

1.3 Code change

Now the register route should be changed like this:

```
@app.post("/register", status_code=201)
async def create_user(user: UserSignIn):
    user.password = pwd_context.hash(user.password)
    q = users.insert().values(**user.dict())
    id_ = await database.execute(q)
    user_do = await database.fetch_one(users.select().where(users.c.id == id_))
    token = create_access_token(user_do)
    return {"token": token}
```

Instead returning the newly created user, we are going to return the token. It depends on the business logic – we can return the token on register and on login or only on login. If let's say we want to let the user enters the application directly after registration, we will return the token, otherwise we will redirect him/her to login page and after successful login we will return the token.

Now if you make a request, the response would be:

```
{
    "token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJleHAiOjE2MzQwMzczMDcsInN1YiI6N30.z2hHbM6i2Xno7pft
8A4pWAgkiDGN1Fbtk4t6qZfwW8E"
}
```

(The token value will be different for you)

2. Authentication

2.1 Custom HTTPBearer class

Let's assume we have a business logic that involves only authenticated users to access the clothes. Since we are doing REST API we could utilize a class called HTTPBearer.
It has a simple definition in the source code like this:

```
class HTTPBearer(HTTPBase):
    def __init__(
        self,
        *,
        bearerFormat: Optional[str] = None,
        scheme_name: Optional[str] = None,
        description: Optional[str] = None,
        auto_error: bool = True,
    ):
        self.model = HTTPBearerModel(bearerFormat=bearerFormat,
description=description)
        self.scheme_name = scheme_name or self.__class__.__name__
        self.auto_error = auto_error

    async def __call__(
        self, request: Request
    ) -> Optional[HTTPAuthorizationCredentials]:
        authorization: str = request.headers.get("Authorization")
        scheme, credentials = get_authorization_scheme_param(authorization)
        if not (authorization and scheme and credentials):
            if self.auto_error:
                raise HTTPException(
                    status_code=HTTP_403_FORBIDDEN, detail="Not authenticated"
                )
            else:
                return None
        if scheme.lower() != "bearer":
            if self.auto_error:
                raise HTTPException(
                    status_code=HTTP_403_FORBIDDEN,
                    detail="Invalid authentication credentials",
                )
            else:
                return None
        return HTTPAuthorizationCredentials(scheme=scheme,
credentials=credentials)
```

We would like to extend this class in our code, because we want to add additional functionality. So instead of copy-paste the code above, let's use the power of inheritance:

```
from fastapi.security import HTTPBearer, HTTPAuthorizationCredentials


class CustomHTTPBearer(HTTPBearer):
    async def __call__(
            self, request: Request
    ) -> Optional[HTTPAuthorizationCredentials]:
        res = await super().__call__(request)

        try:
            payload = jwt.decode(res.credentials, config('JWT_SECRET'),
algorithms=['HS256'])
            user = await database.fetch_one(users.select().where(users.c.id ==
payload["sub"]))
            request.state.user = user
            return payload
        except ExpiredSignatureError:
            raise HTTPException(401, "Token expired.")
        except InvalidTokenError:
            raise HTTPException(401, "Invalid token.")
```

We are overwriting the __call__ method of the base class (HTTPBearer) and first we are calling the super: res = await super().__call__(request)
Then we are storing the result in *res* variable (it you take a look at the base __call__ method it returns HTTPAuthorizationCredentials(scheme=scheme, credentials=credentials) object).

Then we would like to try to decode this token (It comes from the requests header "Authorization") using again the jwt package – this time with decode method. Please, note how we are using a list of algorithms for this function and not only a single algorithm.

If the token is decoded we are fetching the user with the respective id from the database (remember we encoded the id of the user in *sub*?) and we are using a very powerful approach – we are binding to the Request's state a new property called *user* with the value of our already fetched from the database user. This is really useful, because we can inject the request wherever we need, which means now we have a global user access.
Keep in mind this is valid for the current request-response cycle. If you make a request as another user with this request-response cycle it will be the user that you are making a request as.

2.2 Usage

So far so good, but how we can take advantage of all of these

```
oauth2_scheme = CustomHTTPBearer()
```

First, we are creating an object from our custom class *oauth2_scheme*.
Then we are going to create a route for all clothes:

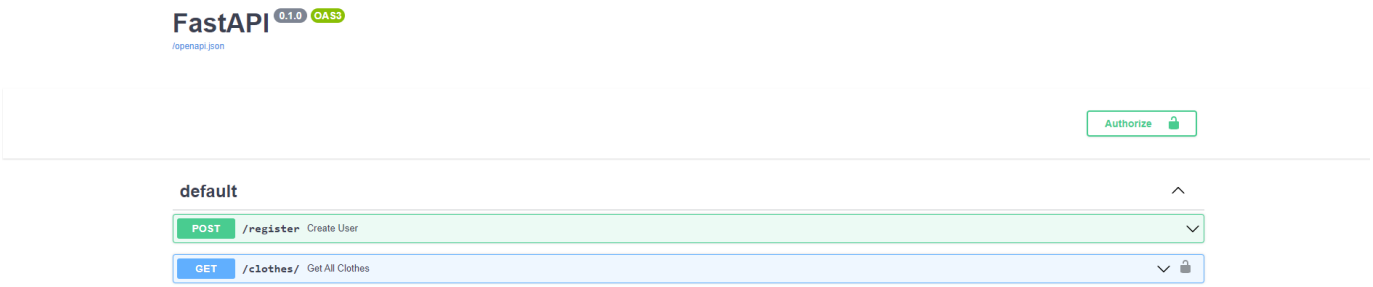
```
@app.get("/clothes/")
async def get_all_clothes():
    return await database.fetch_all(clothes.select())
```

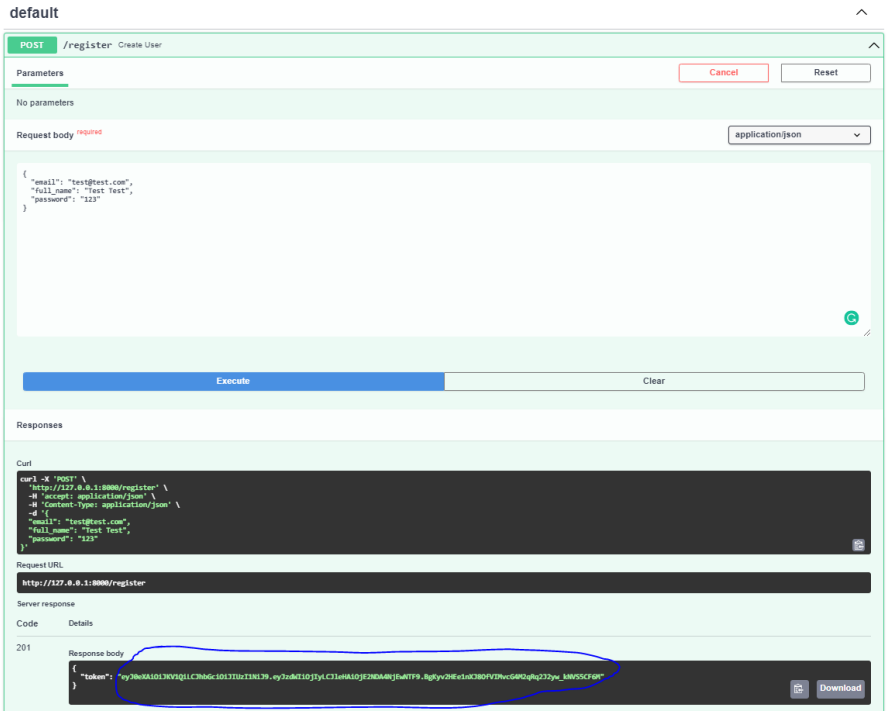
For now, this route is publicly accessible. But we want only authenticated (logged in users) to be able to access it.
We have to use another argument called dependencies:

```
@app.get("/clothes/", dependencies=[Depends(oauth2_scheme)])
async def get_all_clothes():
    return await database.fetch_all(clothes.select())
```

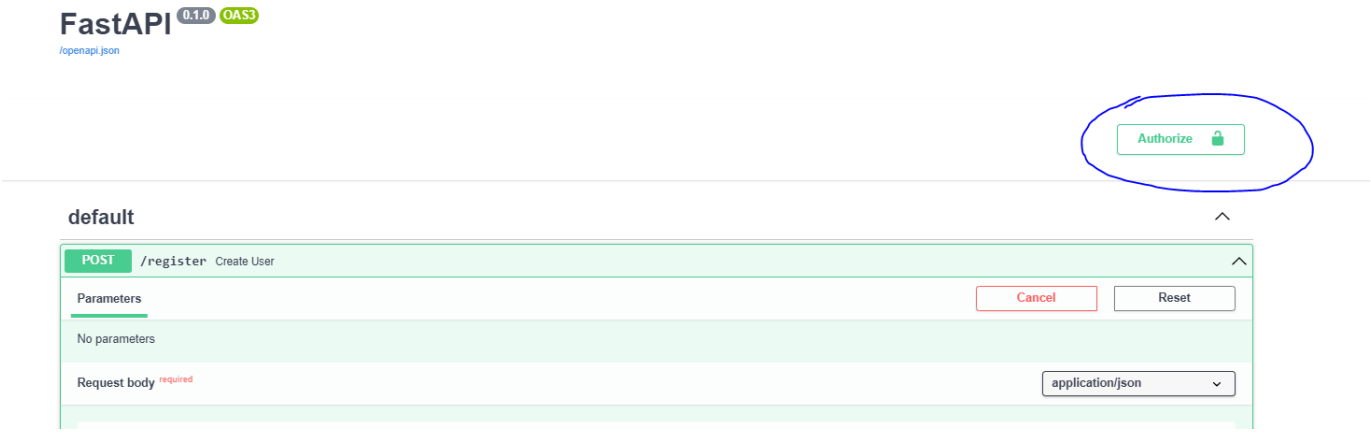
This is a list, which accepts object of type *Depends*. Please, take a closer look and see that we are passing our object from our custom class.

If you start the server and go to the swagger http://127.0.0.1:8000/docs you can make a request to register a user, take only the value (the token) and put it in the authorize button in the top right:



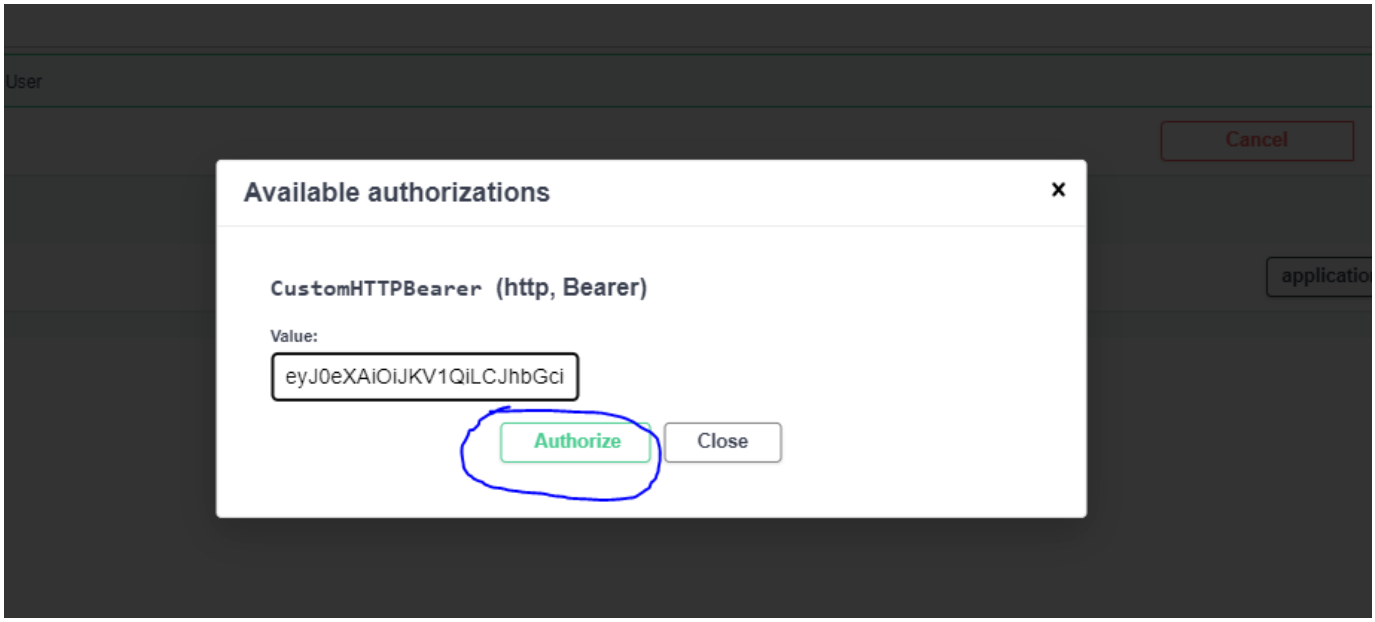


Please copy the value inside the string.

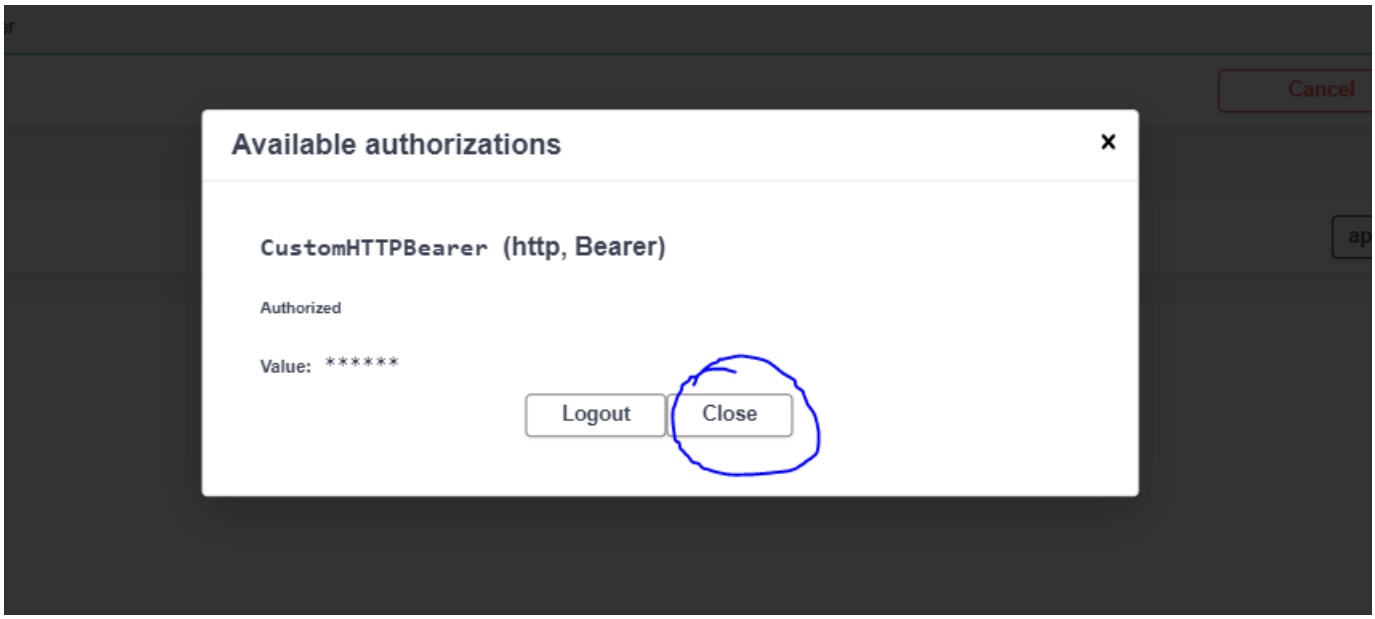


Copy-paste your token in the Authorize button:

CODE WITH FINESSE

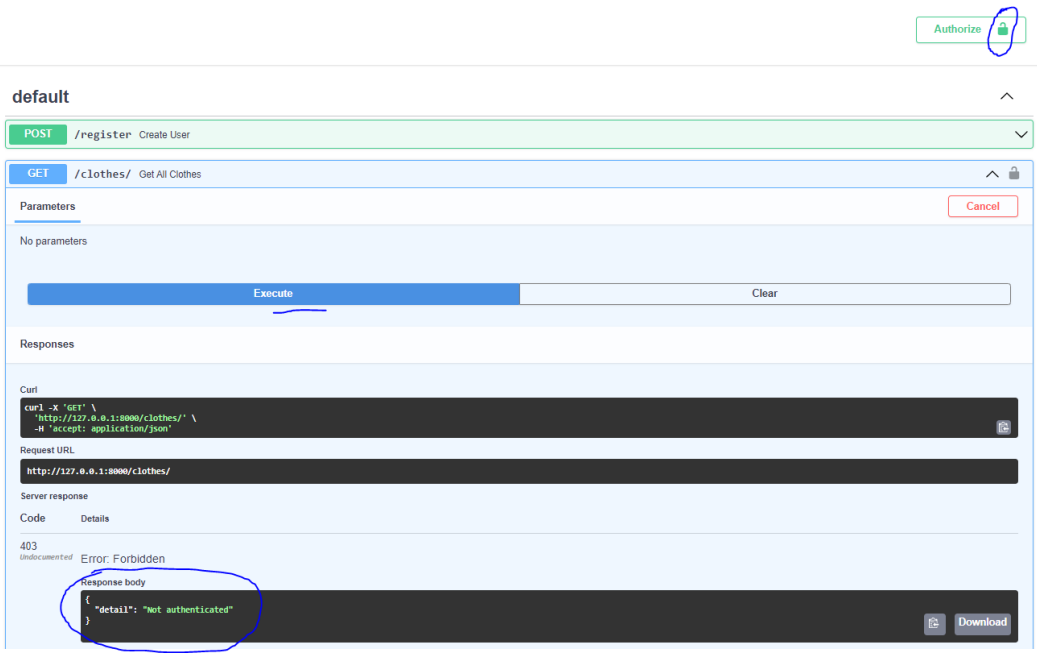


And then click “close”:



If you take a look at the button again you will see now it is locked:

Now if you try the /clothes/ endpoint you will see it is working.
If you want to be 100% sure it is working correctly, go again to “Authorize” and click “Logout”, try again the endpoint /clothes/ and you will receive the following:



Please, debug everything one more time, just to be sure, that you understand how exactly it is working.

NB! Keep in mind that each time you refresh the page http://127.0.0.1:8000/docs you need to register/login again and place the token in the “Authorize button”

3. Authorization

3.1 Change the user model

Let's assume we have a business logic that involves only authenticated **admin** users to create new clothes records in the database.
We need to add new column called ‘role’ and the model would look like this:

```
class UserRolesEnum(enum.Enum):
    super_admin = "super admin"
    admin = "admin"
    user = "user"


users = sqlalchemy.Table(
    "users",
    metadata,
    sqlalchemy.Column("id", sqlalchemy.Integer, primary_key=True),
    sqlalchemy.Column("email", sqlalchemy.String(120), unique=True),
    sqlalchemy.Column("password", sqlalchemy.String(255)),
    sqlalchemy.Column("full_name", sqlalchemy.String(200)),
    sqlalchemy.Column("phone", sqlalchemy.String(13)),
    sqlalchemy.Column("created_at", sqlalchemy.DateTime, nullable=False,
server_default=sqlalchemy.func.now()),
    sqlalchemy.Column(
        "last_modified_at",
        sqlalchemy.DateTime,
        nullable=False,
        server_default=sqlalchemy.func.now(),
        onupdate=sqlalchemy.func.now(),
    ),
    sqlalchemy.Column("role", sqlalchemy.Enum(UserRolesEnum), nullable=False,
server_default=UserRolesEnum.user.name),
)
```

Please, do not forget to create revision and upgrade with alembic.

If for some reason the enum is not created with the migrations and you are receiving message like:
…
sqlalchemy.exc.ProgrammingError: (psycopg2.errors.UndefinedObject) type "userrole" does not exist
LINE 1: ALTER TABLE users ADD COLUMN role userrole DEFAULT 'user' NO...
^

[SQL: ALTER TABLE users ADD COLUMN role userrole DEFAULT 'user' NOT NULL]
(Background on this error at: https://sqlalche.me/e/14/f405)

You can adjust the migration manually:

```
def upgrade():
    # ### commands auto generated by Alembic - please adjust! ###
    user_role = postgresql.ENUM('super_admin', 'admin', 'user', name="user_role")
    user_role.create(op.get_bind())
    op.add_column('users', sa.Column('role', sa.Enum('super_admin', 'admin',
'user', name='user_role'), server_default='user', nullable=False))
    # ### end Alembic commands ###
```

3.2 Extend dependencies

Having only the role column in model would not help us to protect the endpoint.
We need to create s simple function to check if this user has a role ‘admin’:

```
async def is_admin(request: Request):
    user = request.state.user
    if not user or user["role"] not in (UserRolesEnum.admin,
UserRolesEnum.super_admin):
            raise HTTPException(403, "You do not have permissions for this
resource")
```

We need the request injection here, because we can access the user from the request (we did that earlier in the custom HTTPBearer class)

And now we will do something like this, please keep in mind the order is important, first we verify that the user is authenticated (oauth2_scheme – we bind the user to the request there as well) and then we check for the role (if needed):

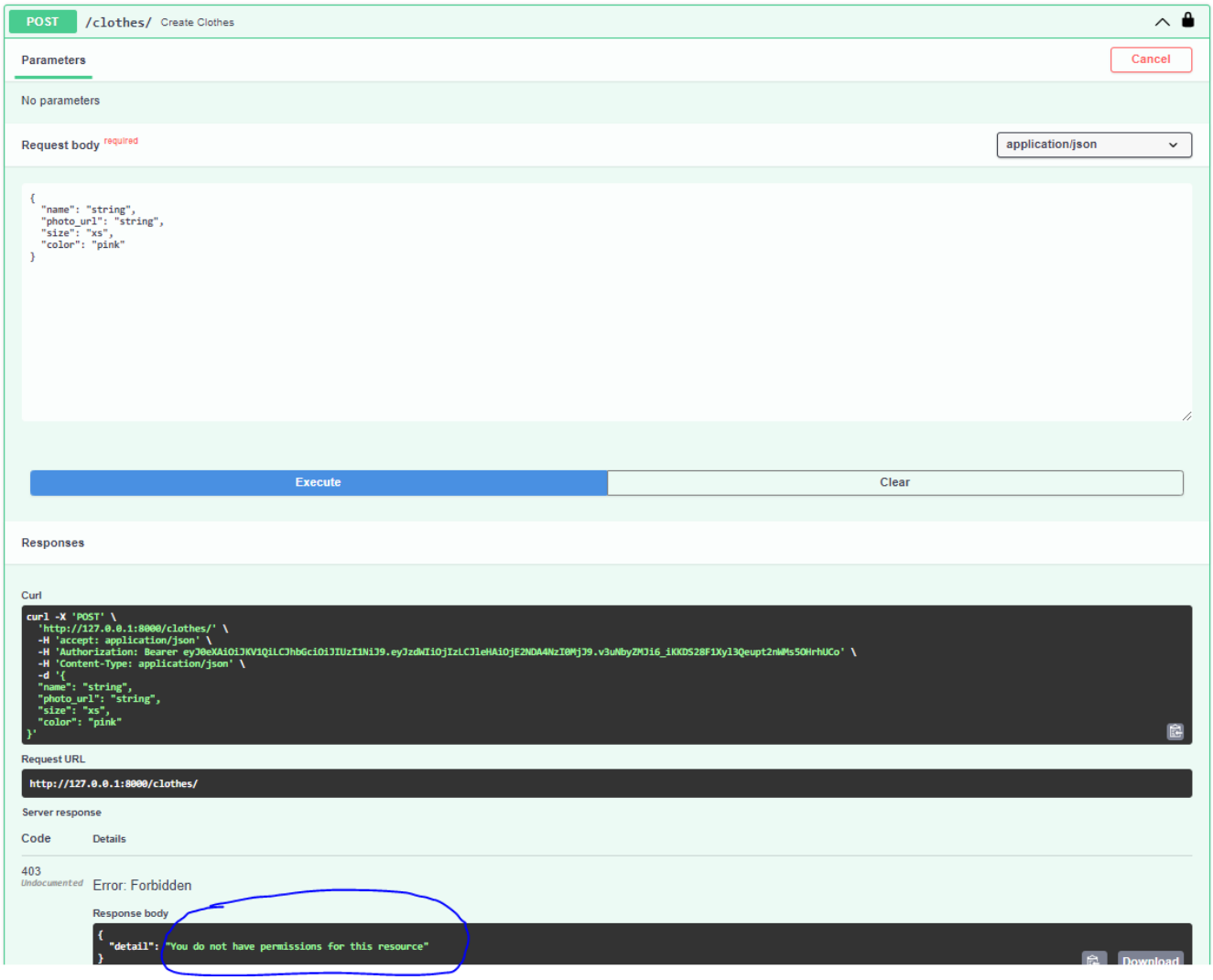
```python
class ClothesBase(BaseModel):
    name: str
    photo_url: str
    size: SizeEnum
    color: ColorEnum


class ClothesIn(ClothesBase):
    pass


class ClothesOut(ClothesBase):
    id: int
    created_at: datetime
    last_modified_at: datetime


@app.post("/clothes/", response_model=ClothesOut, status_code=201,
dependencies=[Depends(oauth2_scheme), Depends(is_admin)])
async def create_clothes(clothes_data: ClothesIn):
    id_ = await database.execute(clothes.insert().values(**clothes_data.dict()))
    return await database.fetch_one(clothes.select().where(users.c.id == id_))
```

Here we create a new endpoint which can create new record for “clothes” table. We define In and Out schema, so our data can be validated and shaped and here
dependencies=[Depends(oauth2_scheme), Depends(is_admin)]
we actually specify that we need not only authentication, but authorization as well.
Now if you try the new endpoint with regular user, you will see this error:



If you remove the token from “Authorize” button then the error will be “Not authenticated”, but with this error code (403 Forbidden) we say- you are authenticated, but you are not authorized to access this resource, because (in our case) you do not have admin role.

In conclusion, we can say that we have done a pretty solid job, but our code is a mess. There are a lot of decorators, models, enumerators, and schemas in one place, and even though the app has only three routes, it is already hard to read.
Do not worry! In the following lecture, we will learn how to structure the Flask app, implement SOLID and MVC principles, and, most importantly, make the code more readable and reusable.