



Search



Get unlimited access to the best of Medium for less than \$1/week. [Become a member](#)



FastAPI Role Base Access Control With JWT



Hirusha Fernando · [Follow](#)

Published in Stackademic

7 min read · Jan 19, 2024



Listen



Share

... More

FastAPI + JWT Role Base Access Control



FastAPI is a modern, high-performance, web framework used to build APIs with Python 3.8+. It is one of the fastest Python frameworks available. In every framework, authentication and authorization are important sections of an API. In this article let's implement Role-based access control with JWT in FastAPI.

Prerequisites

- Python programming knowledge

- Basic knowledge about FastAPI

Before you start you have to install these python modules.

- fastapi
- pydantic
- uvicorn[standard]
- passlib[bcrypt]
- python-jose[cryptography]

Setting Up The Environment

Let's create two API endpoints in the main.py file.

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/hello")
def hello_func():
    return "Hello World"

@app.get("/data")
def get_data():
    return {"data": "This is important data"}
```

Let's create a User Model and Token Model in models.py

```
from pydantic import BaseModel

class User(BaseModel):
    username: str | None = None
    email: str | None = None
    role: str | None = None
    disabled: bool | None = None
    hashed_password: str | None = None

class Token(BaseModel):
    access_token: str | None = None
```

```
refresh_token: str | None = None
```

For this tutorial, I will create a Python dictionary containing dummy users in `data.py`. Also, I will create another list for store refresh tokens. You can use any database for this like PostgreSQL, MongoDB, etc.

```
fake_user_db = [  
    {  
        "username": "johndoe",  
        "email": "john@emaik.com",  
        "role": "admin",  
        "hashed_password": "hdjsbdvdhxbzbsksjdbdbzjdhh45tbdbd7bdbd",  
        "is_active": True  
    },  
    {  
        "username": "alice",  
        "email": "al8ce@emaik.com",  
        "role": "user",  
        "hashed_password": "hdjsbdvdhxbzbsksjdbdbzjdhh45tbdbd7bdbd",  
        "is_active": True  
    }  
]  
  
refresh_tokens = []
```

How This Works

Ok. Now we are ready to implement authentication to our API. Let's start. This is the process. Users should log in using a username and password. This is a post request. Next, the backend validates the user and creates an access token and request token. An access token has a short lifetime. Refresh token has a long lifetime. If a valid user, the backend will send a response with these two tokens. When sending a request to a protected endpoint, the user should attach this access token as a request header. If the access token expires, the user can get a new access token by sending a request to the backend with a refresh token.

Role Based Access Control (RBAC)

FastAPI provides several ways to deal with security. Here we use the OAuth2 with password flow. (You can get more details from [this link](#).) We do that using the `OAuth2PasswordBearer` class. Also, we use `passlib CryptContext` to hash and

verify passwords.

Let's create auth.py. First, create instances of the above classes.

```
#auth.py
from fastapi.security import OAuth2PasswordBearer
from passlib.context import CryptContext

oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")
pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")
```

We pass the tokenUrl parameter to this class. This parameter contains the URL that the client uses to send the username and password in order to get a token. We haven't created this endpoint yet. But we will create it later.

Now create a method to get the user details from db and another method to authenticate users. This method will check the password.

```
#auth.py
from db import User
from passlib.context import CryptContext

pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")

def get_user(db, username: str):
    if username in db:
        user = db[username]
        return User(**user)

def authenticate_user(fake_db, username: str, password: str):
    user = get_user(fake_db, username)
    if not user:
        return False
    if not pwd_context.verify(plain_password, hashed_password):
        return False
    return user
```

Now let's handle the JWT. To do that create some variables and a method to create JWT token.

```
#auth.py
from jose import JWTError, jwt
from datetime import datetime, timedelta, timezone
from data import refresh_tokens

SECRET_KEY = "hdhfh5jdnb7a9563b93f7099f6f0f4caa6cf63b88e8d3e7"
ALGORITHM = "HS256"
ACCESS_TOKEN_EXPIRE_MINUTES = 20
REFRESH_TOKEN_EXPIRE_MINUTES = 120

def create_token(data: dict, expires_delta: timedelta | None = None):
    to_encode = data.copy()
    if expires_delta:
        expire = datetime.now(timezone.utc) + expires_delta
    else:
        expire = datetime.now(timezone.utc) + timedelta(minutes=15)
    to_encode.update({"exp": expire})
    encoded_jwt = jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)
    return encoded_jwt
```

We pass our data and token lifetime to this method and it returns the JWT token.

After that create a method to get details about the current logged-in user. We pass the token to this method. This method decodes the token and gets the user data from the token. And check the user exists in the DB. If there exists return the user. If not raise an exception

```
#auth.py
from typing import Annotated
from jose import JWTError, jwt
from fastapi import Depends, HTTPException, status

SECRET_KEY = "hdhfh5jdnb7a9563b93f7099f6f0f4caa6cf63b88e8d3e7"
ALGORITHM = "HS256"

async def get_current_user(token: Annotated[str, Depends(oauth2_scheme)]):
    credentials_exception = HTTPException(
        status_code=status.HTTP_401_UNAUTHORIZED,
        detail="Could not validate credentials",
        headers={"WWW-Authenticate": "Bearer"},
```

```
)
try:
    payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
    username: str = payload.get("sub")
    if username is None:
        raise credentials_exception
except JWTError:
    raise credentials_exception
user = get_user(fake_users_db, username=username)
if user is None:
    raise credentials_exception
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
```

Also, we need to create another method to check if the user is disabled or not. If not raise an exception. The above code's **Depends()** means dependency. For example, the `get_current_active_user()` method depends on the `get_current_user()` method. If you debug this code you can see when it comes to the `get_current_active_user()`, function, its dependent method, `get_current_user()` will run before the `get_current_active_user()` method.

Now let's create **RoleChecker** class to check user roles. If the role has enough permission it returns True. If not raise an exception

```
#auth.py
from typing import Annotated
from fastapi import Depends, HTTPException, status

class RoleChecker:
    def __init__(self, allowed_roles):
        self.allowed_roles = allowed_roles

    def __call__(self, user: Annotated[User, Depends(get_current_active_user)]):
        if user.role in self.allowed_roles:
            return True
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
```

```
detail="You don't have enough permissions")
```

We have to create one more method to validate the refresh token. When the access token expires, we have to request our refresh token to get a new access token.

```
#auth.py
from typing import Annotated
from fastapi import Depends, HTTPException, status
from fastapi.security import OAuth2PasswordBearer
from data import refresh_tokens

SECRET_KEY = "hdhfh5jdnb7a9563b93f7099f6f0f4caa6cf63b88e8d3e7"
ALGORITHM = "HS256"

oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")

async def validate_refresh_token(token: Annotated[str, Depends(oauth2_scheme)],
                                credentials_exception = HTTPException(status_code=status.HTTP_401_UNAUTHORIZED)):
    try:
        if token in refresh_tokens:
            payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
            username: str = payload.get("sub")
            role: str = payload.get("role")
            if username is None or role is None:
                raise credentials_exception
        else:
            raise credentials_exception

    except (JWTError, ValidationError):
        raise credentials_exception

    user = get_user(fake_users_db, username=username)

    if user is None:
        raise credentials_exception

    return user, token
```

The final auth.py file looks like this.

```

from fastapi.security import OAuth2PasswordBearer
from passlib.context import CryptContext
from db import User
from jose import JWTError, jwt
from datetime import datetime, timedelta, timezone
from data import refresh_tokens
from typing import Annotated
from fastapi import Depends, HTTPException, status

oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")
pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")

SECRET_KEY = "hdhfh5jdnb7a9563b93f7099f6f0f4caa6cf63b88e8d3e7"
ALGORITHM = "HS256"

def get_user(db, username: str):
    if username in db:
        user = db[username]
        return User(**user)

def authenticate_user(fake_db, username: str, password: str):
    user = get_user(fake_db, username)
    if not user:
        return False
    if not pwd_context.verify(plain_password, hashed_password):
        return False
    return user

def create_token(data: dict, expires_delta: timedelta | None = None):
    to_encode = data.copy()
    if expires_delta:
        expire = datetime.now(timezone.utc) + expires_delta
    else:
        expire = datetime.now(timezone.utc) + timedelta(minutes=15)
    to_encode.update({"exp": expire})
    encoded_jwt = jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)
    return encoded_jwt

async def get_current_user(token: Annotated[str, Depends(oauth2_scheme)]):
    credentials_exception = HTTPException(
        status_code=status.HTTP_401_UNAUTHORIZED,
        detail="Could not validate credentials",
        headers={"WWW-Authenticate": "Bearer"},
    )
    try:

```



```

        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        username: str = payload.get("sub")
        if username is None:
            raise credentials_exception
    except JWTError:
        raise credentials_exception
    user = get_user(fake_users_db, username=username)
    if user is None:
        raise credentials_exception
    return user

async def get_current_active_user(current_user: Annotated[User, Depends(get_current_active_user)]):
    if current_user.disabled:
        raise HTTPException(status_code=400, detail="Inactive user")
    return current_user

async def validate_refresh_token(token: Annotated[str, Depends(oauth2_scheme)]):
    credentials_exception = HTTPException(status_code=status.HTTP_401_UNAUTHORIZED, detail="Invalid refresh token")
    try:
        if token in refresh_tokens:
            payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
            username: str = payload.get("sub")
            role: str = payload.get("role")
            if username is None or role is None:
                raise credentials_exception
        else:
            raise credentials_exception

    except (JWTError, ValidationError):
        raise credentials_exception
    user = get_user(fake_users_db, username=username)
    if user is None:
        raise credentials_exception
    return user, token

class RoleChecker:
    def __init__(self, allowed_roles):
        self.allowed_roles = allowed_roles

    def __call__(self, user: Annotated[User, Depends(get_current_active_user)]):
        if user.role in self.allowed_roles:
            return True
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="You don't have enough permissions")

```

Ok. We created the authentication and authorization parts. Now we can add these to our API endpoints. Before doing that we should create two endpoints. One is login and the other one is for refreshing tokens. Let's go to main.py again.

```
from datetime import timedelta
from typing import Annotated

from fastapi import Depends, FastAPI, HTTPException
from fastapi.security import OAuth2PasswordRequestForm

from auth import create_token, authenticate_user, RoleChecker, get_current_a
from data import fake_users_db, refresh_tokens
from models import User, Token

app = FastAPI()

ACCESS_TOKEN_EXPIRE_MINUTES = 20
REFRESH_TOKEN_EXPIRE_MINUTES = 120

@app.get("/hello")
def hello_func():
    return "Hello World"

@app.get("/data")
def get_data():
    return {"data": "This is important data"}

@app.post("/token")
async def login_for_access_token(form_data: Annotated[OAuth2PasswordRequestForm, Depends()]):
    user = authenticate_user(fake_users_db, form_data.username, form_data.password)
    if not user:
        raise HTTPException(status_code=400, detail="Incorrect username or password")

    access_token_expires = timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)
    refresh_token_expires = timedelta(minutes=REFRESH_TOKEN_EXPIRE_MINUTES)

    access_token = create_token(data={"sub": user.username, "role": user.role}, expires_delta=access_token_expires)
    refresh_token = create_token(data={"sub": user.username, "role": user.role}, expires_delta=refresh_token_expires)
    refresh_tokens.append(refresh_token)
    return Token(access_token=access_token, refresh_token=refresh_token)

@app.post("/refresh")
async def refresh_access_token(token_data: Annotated[tuple[User, str], Depends(get_current_active_token)]):
    user, token = token_data
    access_token = create_token(data={"sub": user.username, "role": user.role}, expires_delta=ACCESS_TOKEN_EXPIRE_MINUTES)
    refresh_token = create_token(data={"sub": user.username, "role": user.role}, expires_delta=REFRESH_TOKEN_EXPIRE_MINUTES)
```

```
refresh_tokens.remove(token)
refresh_tokens.append(refresh_token)
return Token(access_token=access_token, refresh_token=refresh_token)
```

Add RBAC To API

Now let's add RBAC to our endpoints. For now, the "/data" endpoint is not protected. It can be accessed by anyone. You can check it using Swagger Docs or Postman. Now let's add RBAC to this endpoint.

```
@app.get("/data")
def get_data(_: Annotated[bool, Depends(RoleChecker(allowed_roles=["admin"]))]
    return {"data": "This is important data"}
```

After doing this, it can be only accessed after login as an admin user. Like that you can add this to any endpoint that you want to protect. Now you know how to add RBAC to FastAPI. This is only one method. There are some other methods to do this. You can find it on the Internet.



Stackademic

Thank you for reading until the end. Before you go:

- Please consider **clapping** and **following** the writer! 🙌
- Follow us [X](#) | [LinkedIn](#) | [YouTube](#) | [Discord](#)
- Visit our other platforms: [In Plain English](#) | [CoFeed](#) | [Venture](#)

[Fastapi](#)[Role Based Access Control](#)[Api Security](#)[Authentication](#)[Web Development](#)



Follow


Written by Hirusha Fernando

79 Followers · Writer for Stackademic

AI Undergraduate | Web Developer | Tech Lover

More from Hirusha Fernando and Stackademic





 Hirusha Fernando in Stackademic

8 Useful Angular Libraries


Hi all 🙌, In this article let's talk about some Angular libraries, useful to your Angular project. You may have heard about some of these...

5 min read · Aug 27, 2023

 510  3




 Oliver Foster in Stackademic

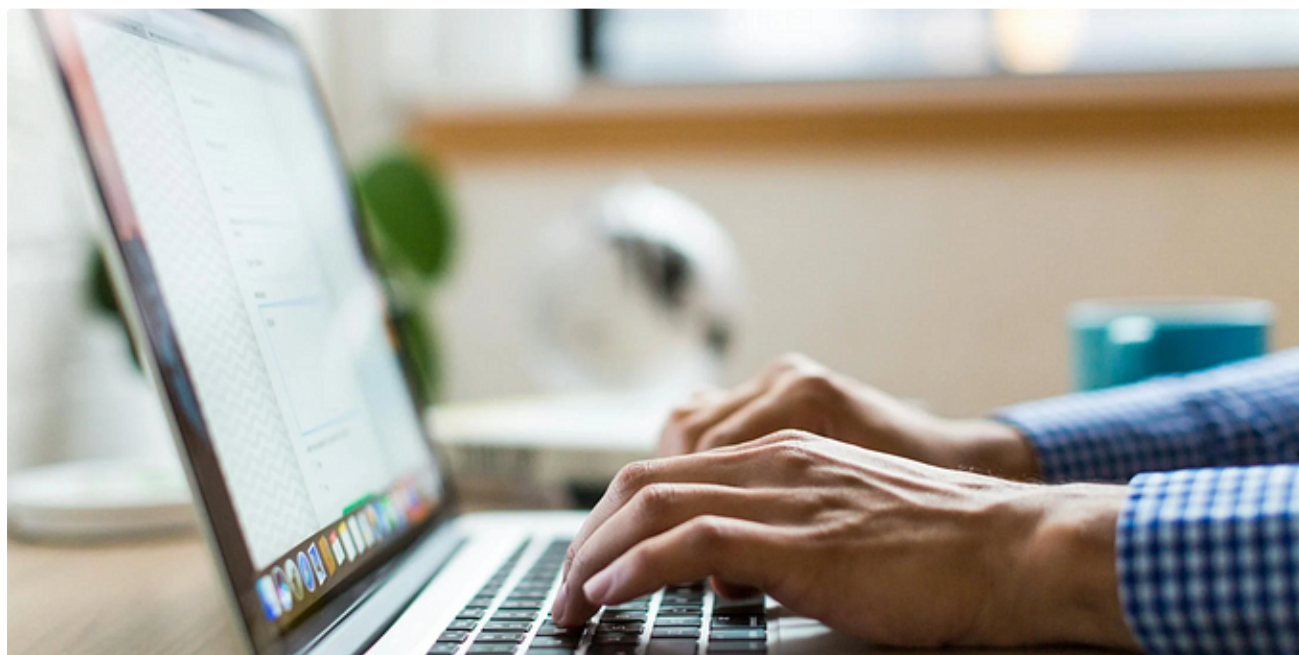
What's the Difference Between localhost and 127.0.0.1?


My article is open to everyone; non-member readers can click this link to read the full text.

★ · 8 min read · Feb 1, 2024

 2.5K  11




 SQL Fundamentals in Stackademic

20 Advanced SQL Techniques

Mastering SQL with Practical Examples

★ · 4 min read · Feb 17, 2024




 Hirusha Fernando in Stackademic

PII Masking With Python Using ai4privacy

PII (Personal Identifiable Information) masking is a technique used to hide sensitive data from public. We can use machine learning model...

4 min read · Feb 29, 2024

 8   

See all from Hirusha Fernando

See all from Stackademic

Recommended from Medium



Merwan Chinta in CodeNx

Crafting with FastAPI, SQLAlchemy and Pydantic

In this article, we explore the creation of a dynamic product catalog using FastAPI, SQLAlchemy, and Pydantic.

4 min read · Feb 21, 2024



118



message
in Login
ies Get Movies
ies Create Movie
ies/{id} Get Movie
ies/{id} Update Movie
ies/{id} Delete Movie

 Jordi Oltra

Basic FastAPI good practices

How works a basic API

10 min read · Dec 3, 2023

 95



Lists



Coding & Development

11 stories · 519 saves



General Coding Knowledge

20 stories · 1040 saves



Tech & Tools


16 stories · 189 saves



Stories to Help You Grow as a Software Developer

19 stories · 922 saves



 Thomas Aitken

Setting up a FastAPI App with Async SQLALchemy 2.0 & Pydantic V2

Early this year, a major update was made to SQLAlchemy with the release of SQLAlchemy 2.0. Among other things, this includes significant...

11 min read · Oct 20, 2023



293



3



Building Fast and Robust APIs with FastAPI and SQL Databases

@niroshanyi



Yasantha Niroshan in Towards Data Engineering

Building Fast and Robust APIs with FastAPI and SQL Databases

In the realm of web development, the creation of fast and dependable APIs is an essential aspect. One noteworthy and modern Python web...

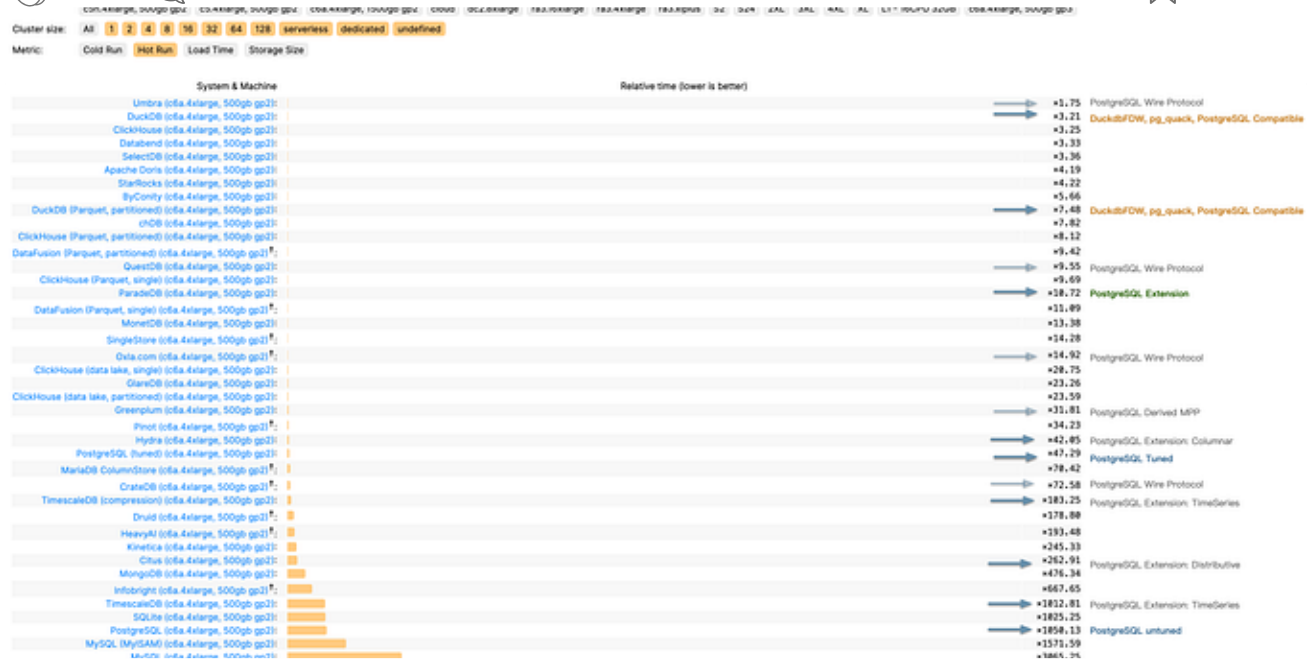
5 min read · Sep 29, 2023



203



1



Vonng

Postgres is eating the database world

PostgreSQL isn't just a simple relational database; it's a data management framework with the potential to engulf the entire database...

11 min read · Mar 15, 2024



1.2K



12





DavidW (skyDragon) in overcast blog

13 Docker Tricks You Didn't Know

Docker has become an indispensable tool in the development, testing, and deployment pipelines of modern applications. While many Docker...

21 min read · Mar 14, 2024



705



1



See more recommendations