**Enhancing Microservice Security with Custom Middleware in Django**

[[elijah samson](https://medium.com/@elijahobara?source=post_page-----e25409c45e94--------------------------------)](https://medium.com/@elijahobara?source=post_page-----e25409c45e94--------------------------------)

[[AWS Tip](https://awstip.com/?source=post_page-----e25409c45e94--------------------------------)](https://awstip.com/?source=post_page-----e25409c45e94--------------------------------)

[elijah samson](https://medium.com/@elijahobara?source=post_page-----e25409c45e94--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2Fd5d124924232&operation=register&redirect=https%3A%2F%2Fawstip.com%2Fenhancing-microservice-security-with-custom-middleware-in-django-e25409c45e94&user=elijah+samson&userId=d5d124924232&source=post_page-d5d124924232----e25409c45e94---------------------post_header-----------)

Published in

[AWS Tip](https://awstip.com/?source=post_page-----e25409c45e94--------------------------------)

·

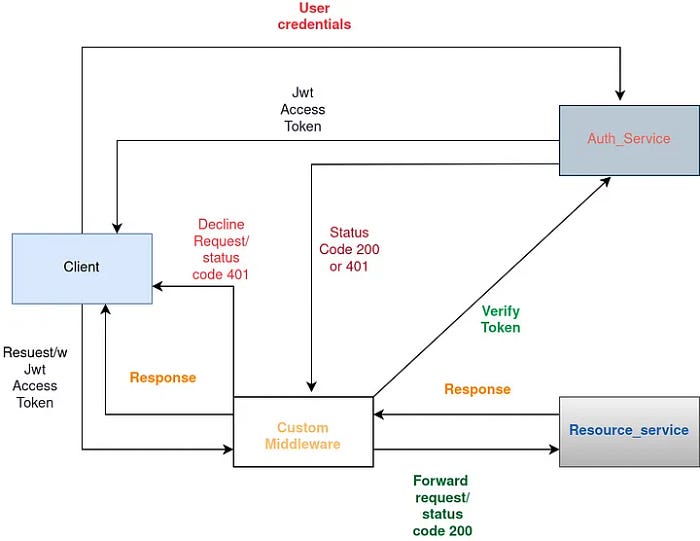
6 min read

·

Oct 22

81

1



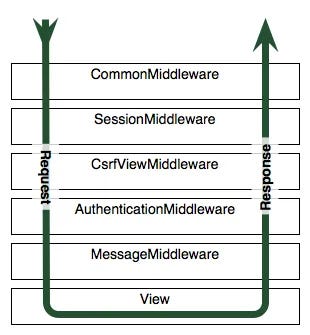
Today i will discuss about Microservice Authentication using Django’s custom middleware.Before delving into the topic it is important to understand middleware in django.

**What is Django middleware?**

***Middleware*** is a fundamental component that can intercept and modify HTTP requests and/or responses as they flow through the Django application. **Django middleware is designed to be chained together, forming a pipeline of behavioral changes during the request processing lifecycle.** Each middleware class can perform specific tasks or add functionality to the request or response without being solely responsible for generating the final response to the client.

Examples of common Django middleware tasks include request logging, authentication, CORS handling, caching, and session management. Each middleware class in the pipeline can alter the request or response in some way, and the modified request/response then passes on to the next middleware in the sequence. This sequential processing allows developers to apply different behaviors and transformations to the request and response at various stages of the pipeline.

In Django,**the order of middleware classes in the MIDDLEWARE setting matters as it defines the sequence of their execution. The first middleware in the list is the first to receive the incoming request, and the last middleware sends the response back to the client.**



Request-Response cycle

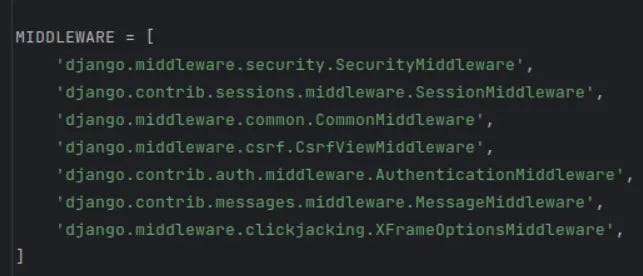
When a request comes then it executes layer by layer middleware classes, once the request is processed by the view and response is returned, then it would execute every middleware class opposite order. The above image depicts the request-response cycle and how Django middleware is executed.

There are two types of Middleware in Django.

* ***Built-in Middleware***
* ***Custom Middleware***

1. **Built-in Middleware**

When we install Django by default Django will add the following middleware in ***settings.py***.



**2. Custom Middleware**

We can write our own ***Custom Middleware***. Custom Middleware can be ***Function-based*** or ***Class-based.***

Now, let’s explore how we can leverage this custom middleware to implement authentication in our microservice.

Now, let’s create two microservices: auth\_service and test\_service.

***auth\_service***and ***test\_service*** are ***REST*** applications and they will communicate with each other using ***REST APIs***.We will leverage the power of ***django-rest-framework*** combined with ***djangorestframework-simplejwt***.

By using ***simplejwt***, we can establish a robust JWT-token manager that handles user authentication, token generation upon request, verification of token validity and expiration, and the ability to refresh tokens as needed.

In the ***urls.py*** file of ***auth\_service***, include the views for creating and verifying JWT Token.

from django.urls import path  
from rest\_framework\_simplejwt import views as jwt\_views  
  
urlpatterns = [  
 path('token/', jwt\_views.TokenObtainPairView.as\_view(), name='token\_obtain\_pair'),  
 path('verify/', jwt\_views.TokenVerifyView.as\_view(), name='verify'),  
]

[**http://127.0.0.1:8080/auth/api/token/**](http://127.0.0.1:8080/auth/api/token/)**.**This URL is the authentication endpoint responsible for issuing access tokens and refresh token.

[**http://127.0.0.1:8080/auth/api/verify/**](http://127.0.0.1:8080/auth/api/verify/)**.**This URL is the verification endpoint used to validate and verify the authenticity of an access token.

For simplicity we can register a user using the **python3 manage.py createsuperuser** command.

In the ***views.py*** file of the ***test\_service***, include the provided test\_api view function.

from django.http import JsonResponse  
from rest\_framework.decorators import api\_view  
  
@api\_view(['GET'])  
def test\_api(request):  
 data = {  
 'message': 'Hello, this is a test API!',  
 'data': {  
 'example\_data': 123456,  
 }  
 }  
 return JsonResponse(data)

The URL endpoint of this ***test\_api*** view function is **“**[**http://127.0.0.1:8081/api/test/**](http://127.0.0.1:8081/api/test/)**".**

Now create a new python file name it ***middleware.py*** and place it in ***test\_service*** directories.In the ***middleware.py***, we can define our middleware class.A middleware class should have an **\_\_init\_\_** method and **\_\_call\_\_** method.

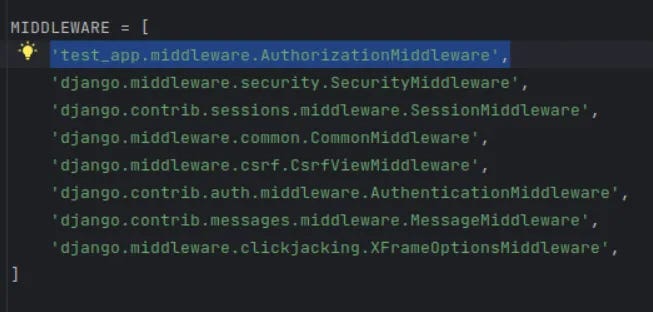
import requests  
from django.http import JsonResponse  
  
  
class AuthorizationMiddleware:  
 def \_\_init\_\_(self, get\_response):  
 self.get\_response = get\_response  
  
 def \_\_call\_\_(self, request):  
 token = request.META.get('HTTP\_AUTHORIZATION', None)  
  
 if token:  
 verification\_url = "http://127.0.0.1:8080/auth/api/verify/"  
  
 response = requests.post(verification\_url, data={'token': token})  
  
 if response.status\_code == 200:  
  
 return self.get\_response(request)  
 else:  
  
 return JsonResponse({'error': 'Unauthorized'}, status=401)  
  
 return JsonResponse({'error': 'Token not found'}, status=401)

***AuthorizationMiddleware*** responsible for handling token-based authentication for incoming requests.The ***AuthorizationMiddleware***class is initialized with the ***get\_response*** argument, which represents the next middleware or view in the request-response pipeline.

The ***\_\_call\_\_***method is the core of the middleware and gets called for each incoming request.The middleware first extracts the token from the request’s ***HTTP\_AUTHORIZATION*** header using ***request.META.get(‘HTTP\_AUTHORIZATION’, None)***.

If a token is found in the request headers, it proceeds with the following steps:It sends a POST request to the verification\_url [**http://127.0.0.1:8080/auth/api/verify/**](http://127.0.0.1:8080/auth/api/verify/) with the extracted token in the request data (payload). The purpose of this request is to verify the token’s authenticity and validity. If the verification\_url returns a successful response with a status code of ***200***, it means the token is valid, and the middleware allows the request to proceed to the next middleware or view in the pipeline by calling ***self.get\_response(request).***If the verification\_url returns an error response with a status code other than ***200***, it indicates that the token is either invalid, expired, or there’s an authentication issue. In this case, the middleware returns a JSON response with an error message indicating “***Unauthorized***” and a status code of ***401*** (Unauthorized).

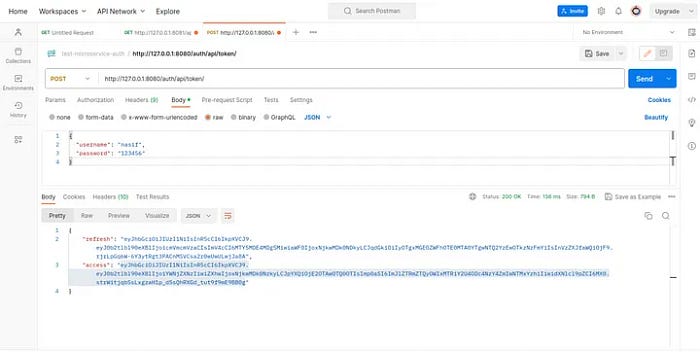
Now we will need to add it to the ***MIDDLEWARE*** setting in our ***test\_service*** Django settings.



We can write the middleware as a ***function-based***middleware as well.

import requests  
from django.http import JsonResponse  
  
def AuthorizationMiddleware(get\_response):  
 def middleware(request):  
 token = request.META.get('HTTP\_AUTHORIZATION', None)  
 if token:  
 verification\_url = "http://127.0.0.1:8080/auth/api/verify/"  
 response = requests.post(verification\_url, data={'token': token})  
 if response.status\_code == 200:  
 return get\_response(request)  
 else:  
 return JsonResponse({'error': 'Unauthorized'}, status=401)  
 return JsonResponse({'error': 'Token not found'}, status=401)  
 return middleware

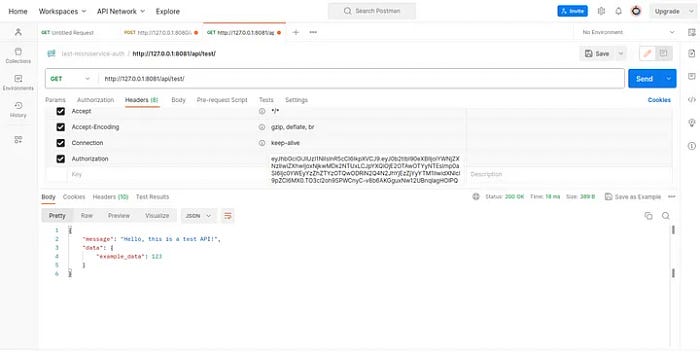
Now, let’s proceed to test our ***AuthorizationMiddleware*** using Postman.



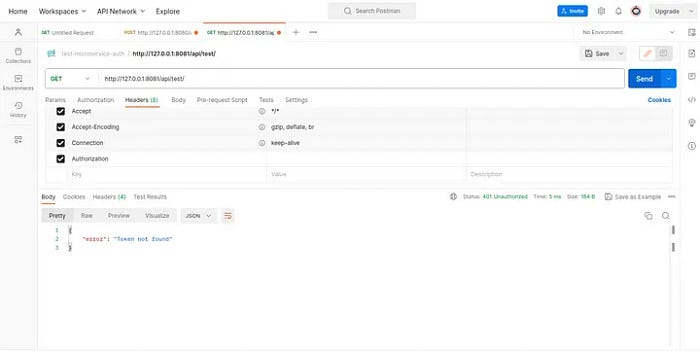
To obtain the access and refresh tokens from the **“**[**http://127.0.0.1:8080/auth/api/token/**](http://127.0.0.1:8080/auth/api/token/)**"** endpoint using Postman,Set the request type to “POST” and Add two key-value pairs in the body:

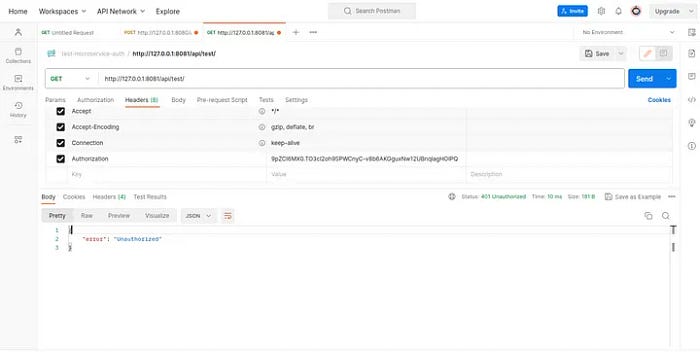
* Key: **username**, Value: superuser\_username
* Key: **password**, Value: superuser\_password

Now copy the access token value.Open a new request tab in Postman.Set the request type to “GET”. Enter the URL **“**[**http://127.0.0.1:8081/api/test/**](http://127.0.0.1:8081/api/test/)**"** in the address bar.In the “Headers” section of the request, add a new header with the key “***Authorization***” and the value access token obtained from the previous request.



Without token or invalid token it will give status code of ***401*** (Unauthorized).





If we have multiple services and want to avoid adding the middleware.py file in every service, we can create a ***Python package*** containing the middleware logic. This package can be installed in each microservice as a dependency, allowing them to have the ability to communicate with the ***auth\_service***.This will ensure code reusability across multiple services.

**Is it possible to authenticate our services without relying on communication with the auth\_service?**

**Yes**,it is possible.To achieve this, the same secret key needs to be set in the ***‘SIGNING\_KEY’***configuration for both the ***auth\_service*** and the other service.In this setup, the responsibility of token validation lies with the individual services, not the central auth\_service. The advantage of this approach is that even if the auth\_service becomes unavailable due to any reason, all other services in the API will still be able to function independently, as they can locally validate the tokens without relying on auth-service.

That’s it, I guess.

Cheers, Happy Coding!!!

🙏🙏🙏

Since you’ve made it this far, sharing this article on your favorite social media network would be highly appreciated. For feedback,[please ping me on Twitter.](https://twitter.com/talkcoding)