Token Authentication in Django Channels and Websockets

Last Updated : 24 Sep, 2024

Prerequisites: Django, WebSockets, Django channels, Token authentication

The most popular Django topics right now are WebSockets and Django channels because they make it possible to communicate quickly and easily while working in real-time without constantly refreshing the page. When working with the Django templating engine alone and not the REST framework, the authentication method that Django channels employ is session authentication. The user who is currently logged in is also reflected in the Django channels auth system.

```
// This is not possible in websockets .

var socket = new WebSocket("path_of_the_socket" , headers = {
"Authorization': "token token_string" })
```

Well, using web-based authentication is one way you could reach your goal. Returning a cookie to the website allows you to connect to the WebSocket server once more with the cookie in hand, authenticate, and then continue. However, cookie-based authentication can be extremely unreliable and have various problems. Therefore, to address this issue, we can convey the authentication token from the socket's query parameters and then build a new authentication stack that will sit on top of the existing ones when routing WebSocket connections.

However, working with the rest framework with token authentication or JWT authentication is not simple for WebSockets and channels; as a result, it can be challenging to log in to the user via channels and sockets when using token authentication or the rest framework. There is no way you can send auth/JWT

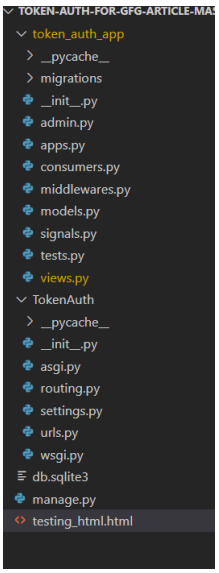
tokens to the headers of a WebSocket like that of a simple post or get a request in the Authorization header.

Note: This process is totally asynchronous because WebSockets and channels work with ASGI (Asynchronous server gateway interface).

Setting up our Project

File Structure

This is the file structure after completing all the steps in this article.



Step 1: Create a virtual environment and activate it.

Step 2: Install Django and start your project.

Step 3: Create a project named TokenAuth

```
django-admin startproject TokenAuth
```

Step 4: Start an app named token_auth_app by moving to the project folder.

```
cd TokenAuth
python manage.py startapp token_auth_app
```

Step 5: Add your app to the installed apps in setting.py.

```
'token_auth_app.apps.TokenAuthAppConfig'
```

Step 6: To Set up Django channels for the backend part. Install Django-channels

```
python -m pip install -U channels
```

Step 7: Add channels to the installed apps.

Python

```python
INSTALLED_APPS = [
    'token_auth_app.apps.TokenAuthAppConfig',
    'channels'
]
```

Step 8: Installing the REST framework is required to create token authentication for backend applications. For token authentication, the rest framework's auth token package, which contains a Token model, is used.

```
pip install django-rest-framework
```

Step 9: Install cors.

```
pip install django-cors-headers
```

Step 10: Add both to the INSTALLED_APPS and also token auth package.

Python

```python
INSTALLED_APPS = [
    'token_auth_app.apps.TokenAuthAppConfig',
    ...
    'channels',
    'rest_framework',
    'rest_framework.authtoken',
    'corsheaders',
]
```

Step 11: Set ALLOWED_HOSTS to *, Also, add the cors middleware to the MIDDLEWARE list The cors setup is done so that the host allows all the origins and we can communicate with the backend application.

Python

```python
ALLOWED_HOSTS = ["*"]

CORS_ALLOW_ALL_ORIGINS = True


MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',

    # --------------------------------------
    'corsheaders.middleware.CorsMiddleware',
    # --------------------------------------
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',

'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]
```

Token authentication is essential when working with WebSockets in Django Channels. If you want to explore such advanced Django features more thoroughly, the **Complete Django Web Development Course – Basics to Advance** offers detailed guidance

Code Implementation

Setting up files one by one:

token_auth_app/consumers.py file:

In the app folder create a consumers.py that will deal with all the incoming receive and send commands from the WebSocket

Python

```python
from channels.generic.websocket import
AsyncJsonWebsocketConsumer
```

```
2
3  class TokenAuthConsumer(AsyncJsonWebsocketConsumer):
4      async def connect(self):
5          await self.accept()
6          print(self.scope["user"].username)
7          print(self.scope["user"].email)
8
9      async def disconnect(self, close_code):
10         ...
11
12     async def receive_json(self, message):
13         command = message.get("command")
14         if command == "Say hello !":
15             print(message["data_string"])
16             await self.send_json({
17                 "command_response": "The command to \
18                 say hello was received ",
19                 "data_string_backK": message.get
20               ("data_string", None)
21             })
```

Code explanation:

We have created a Consumer with the class name TokenAuthConcumser as it consumes events sent and received by the ASGI application which we will see create later. Here, we inherit **AsyncJsonWebsocketConsumer**, which is the Parent Consumer class. In the AsyncJsonWebsocketConsumer module, we have send_json and receive_json functions which accept in JSON and send JSON without dumping JSON.

1. **connect function:** The first and foremost function that is executed in order to establish a connection is the one that takes the WS to connect request, thus we must connect it. The scope is the dictionary, which functions similarly to the request parameter in function-based views (def fun(request)) in that it contains all the information about the incoming connection. user is accessed in scope as a dict object scope[“user”], which will provide information on the user who is currently logged in. The user’s username and email are then retrieved and displayed.
2. **disconnect function:** Takes the close_code parameter and then does nothing.

3. **receive_json function:** We must implement the receive method in AsyncWebsocketConsumer. then, in order to retrieve items, we must load the data (json.loads). However, you have direct access to the stuff here. This procedure uses the message sent from the front end. We will send this data as given below:

JavaScript

```
{
    "command" : "Say hello !" ,
    "data_string" : "This is the data string !" ,
}
```

As this data is received, we interpret it and send responses accordingly in the front end.

token_auth_app/middlewares.py file:

It will have the custom auth which we will create and the custom stack will be superposed above all the stacks.

Python

```
from rest_framework.authtoken.models import Token
from urllib.parse import parse_qs
from channels.db import database_sync_to_async
from django.contrib.auth.models import AnonymousUser


@database_sync_to_async
def returnUser(token_string):
    try:
        user = Token.objects.get
            (key=token_string).user
    except:
        user = AnonymousUser()
    return user

```

```
16
17 class TokenAuthMiddleWare:
18     def __init__(self, app):
19         self.app = app
20
21     async def __call__(self, scope, receive, send):
22         query_string = scope["query_string"]
23         query_params = query_string.decode()
24         query_dict = parse_qs(query_params)
25         token = query_dict["token"][0]
26         user = await returnUser(token)
27         scope["user"] = user
28         return await self.app(scope, receive, send)
```

Code explanation:

The uses of different modules are described below:

1. **Token:** This is used for token auth validation.
2. **parse_qs:** It is used for parsing the query parameters from string to dict.
3. **database_sync_to_async:** It is used for retrieving the data from the database as Django ORM is totally synchronous.
4. **AnonymousUser:** This helps if we don't find the user based on the token, i.e. **returnUser** function returns the user related to the token passed to the function else it returns the AnonymousUser.

In the __init__ function we define the current instance app to the app which is passed into the stack. Next, the async __call__() function takes three parameters scope, receive and send. The scope is the dictionary which has all the data about the connection as stated above. send and receive are the commands for sending and receiving commands. Every ASGI application has three parameters environ, send and receive, here environ is the scope var. The query_string is the string passed from the front request, For example:

```
ws://localhost:8000/?token=fkdsahjkfshiadjkfhoasdfk"
```

The query string that is provided in the query string variable in scope is token=fkdsahjkfshiadjkfhoasdfk. It is sent in bytes as a string. This indicates that the query string's type is bytes.

The byte string is converted to a python string using a query string.decode(). p Parse qs(query params), on the other hand, decodes the string and adds it to the Python dictionary. The dict is presented as a key-value pair. Therefore, we access the token key, then access the token's first element, token=qury_dict["token"][0], and finally, we visit the returnUser method to obtain the user. and we set it to the scope's "user" property before returning the application.

The interesting part about this scope variable is that you can completely change it and make any changes you desire. This entails that you are free to change the variables in the scope parameter as well as add new ones.

for adding, just add like a normal Python dictionary, i.e. scope["some_another_key"] = "value_for_the_key",

for deleting, del scope["some_key_that_exist"]

TokenAuth/routing.py file:

It will create the ASGI application which will be served and which will route all the asynchronous WebSocket URLs to the respective consumers.

Python

```python
from channels.routing import ProtocolTypeRouter, URLRouter
from channels.security.websocket import AllowedHostsOriginValidator
from django.urls import path
from token_auth_app.consumers import TokenAuthConsumer
from token_auth_app.middlewares import TokenAuthMiddleWare

application = ProtocolTypeRouter(
    {
        "websocket": TokenAuthMiddleWare(
            AllowedHostsOriginValidator(
                URLRouter(
                [path("", TokenAuthConsumer.as_asgi())]
                )
            )
        )
```

```
        }
    )
```

Code explanation:

The uses of different modules are described below:

1. **ProtocolTypeRouter**: This routes the server of the whole application based on the type of protocol. If the protocol is HTTP it serves the Django WSGI app and if WS then the ASGI app.
2. **URLRouter**: This routes the WS connection routes.
3. **AllowedHostsOriginValidator**: This security stack is used to validate the connection and data acceptance only from the hosts which are allowed in the settings.py file of the Django application.
4. **path**: Routing the path.
5. **TokenAuthConsumer**: To serve the consumer instance for a particular connection.
6. **TokenAuthMiddleware**: For setting user instance using token passed to the connection.

Then, set the stacks for serving the ASGI app in the application variable. Keep in mind that we make use of TokenAuthMiddleWare, and that's where we can set the user using the token. In the token auth app middlewares.py, we recently developed that class.
You may have noticed that we do not use AuthMiddlewareStack. This is because it is helpful when serving the ASGI app from the same application or templating engine. It stores the data in the sessions.

ASGI Application and Channel Layers in settings.py

Set your ASGI_APPLICATION will serve the required consumers and route. Whereas in CHANNEL_LAYERS, we are using InMemoryChannel this is for a development environment.

Python

```
WSGI_APPLICATION = 'TokenAuth.wsgi.application'
```

```
ASGI_APPLICATION = 'TokenAuth.routing.application'


CHANNEL_LAYERS = {
    "default": {
        "BACKEND":
      "channels.layers.InMemoryChannelLayer",
    }
}
```

token_auth_app/signals.py

This file creates a token object for each and every user that will be created and this signal.py will be imported by the apps.py file in the ready function.

Python

```
from django.dispatch import receiver
from django.contrib.auth.models import User
from django.db.models.signals import post_save
from rest_framework.authtoken.models import Token


@receiver(post_save, sender=User)
def create_token(sender, instance, created, **kwargs):
    if created:
        token = Token.objects.create(user=instance)
        token.save()
```

token_auth_app/apps.py

Now after setting all the application's backend, and database, this will add the token model, sessions, and rest framework into work.

Python

```
from django.apps import AppConfig
```

```
2
3
4  class TokenAuthAppConfig(AppConfig):
5      default_auto_field =
6          'django.db.models.BigAutoField'
7      name = 'token_auth_app'
8
9      def ready(self):
10         import token_auth_app.signals
```

token_auth_app/views.py

Here we will create a login page for the frontend part and from there we will make a post request with username and password then the backend view will validate whether the user is there or not, if the user is there then the token will be sent from the backend in the response and we will create a socket connection from there. For this, we will need a views.py which will have will handle login and send the token.

The uses of different modules are described below:

1. **Response**: This is the response object which is sent from the API view
2. **api_view**: It converts the normal Django view to the API view.
3. **authenticate**: This will return the user, based on the username and password.

Python

```
1  from rest_framework.response import Response
2  from rest_framework.decorators import api_view
3  from rest_framework.authtoken.models import Token
4  from django.contrib.auth import authenticate
5
6
7  @api_view(["POST"])
8  def loginView(request, *args, **kwargs):
9      username = request.POST.get("username")
10     password = request.POST.get("password")
11     try:
```

```
        user = authenticate(username=username,
                            password=password)
    except:
        user = None
    if not user:
        return Response({
            "user_not_found": "There is no user \
            with this username and password !"
        })
    token = Token.objects.get(user=user)
    return Response({
        "token": token.key,
    })
```

TokenAuth/urls.py

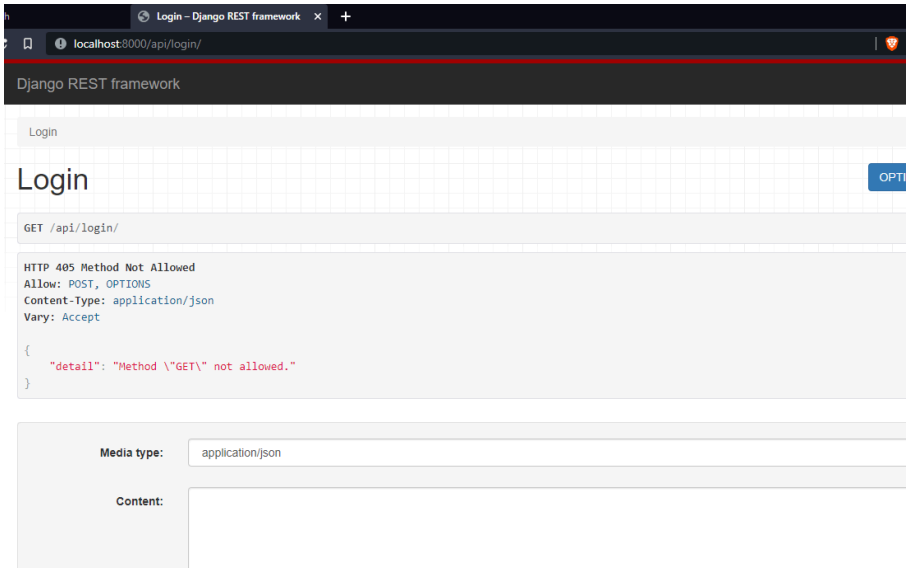
We need to set the URL for this page in Project urls.py add this URL to the loginview with URL route = “api/login/”. This file will handle the backend part for the login and will return a response on the basis of the username and the password.

Python

```python
from django.contrib import admin
from django.urls import path
from token_auth_app.views import loginView

urlpatterns = [
    path('admin/', admin.site.urls),
    path("api/login/", loginView),
]
```

It will look like this:



testing_html.html file

After submitting a post request with a username and password, the file will be used to store the token that was received upon successful authentication. The TokenAuthMiddleWare in middlewares.py will assist us with this. Then, using Django channels, we will connect to a WebSocket to the backend and authenticate using the token.

HTML

```
<!DOCTYPE html>
<html>
    <head>
        <title>Token Auth</title>
    </head>
    <body>
        <input type ="text" name = "username" id = "form_username" /><br>
        <input type = "text" name = "password" id = "form_password" /><br>
        <button onclick = "loginViaBackend()"> Login </button>

        <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.6.0/jquery.min.js"></script>

        <script>
            function createSocket(token) {
```

```
15                 ws_path = "ws://localhost:8000/?
token="+token ;
16                 socket = new WebSocket(ws_path) ;
17
18                 socket.onopen = function(e) {
19                     console.log("Connected") ;
20                     socket.send(
21                         JSON.stringify(
22                             {
23                                 "command" : "Say hello !"
,
24                                 "data_string" : "This is
the data string !" ,
25                             }
26                         )
27                     ) ;
28                 }
29
30                 socket.onclose = function(e) {
31                     console.log(e) ;
32                 }
33
34                 socket.onmessage = function(data) {
35                     data = JSON.parse(data.data) ;
36                     console.log(data) ;
37                     console.log(data["command_response"])
38                 }
39             }
40
41             function loginViaBackend() {
42                 console.log("The function was run !") ;
43                 var username =
document.getElementById("form_username").value ;
44                 var password =
document.getElementById("form_password").value ;
45
46                 console.log(username) ;
47                 console.log(password) ;
48
49                 payload = {
50                     "username" : username ,
51                     "password" : password ,
52                 }
53
```

```
                $.ajax(
                    {
                        "type": "POST" ,
                        "dataType" : "json" ,
                        "url" :
"http://localhost:8000/api/login/" ,
                        "timeout" : 5000 ,
                        "data" : payload ,
                        "success" : function(data) {
                            console.log(data) ;
                            token = data["token"] ;
                            createSocket(token) ;
                        },
                        "error" : function(data) {
                            console.log(data) ;
                        }
                    }
                ) ;
            }
        </script>
    </body>
</html>
```

Code explanation:

Once you click the login button the function loginViaBackend function is run and it makes a post request to the api/login URL which we have just created in the backend. (we are using ajax for making the post request), In the loginViaBackend function, we access the values of the username and the password and then send it via the post request. After the successful request we get the token from the backend as the successful request of a username and the password returns the token.

Create a superuser

```
python manage.py createsuperuser
```

I have created a superuser named, **testuserone** with password **testing123**, and **suraj** with password **123**.

```
((project_1) ) C:\Users\suraj\Downloads\Token-Auth-for-GFG-Article-mas
Username (leave blank to use 'suraj'): suraj
Email address: suraj@gmail.com
Password:
Password (again):
This password is too short. It must contain at least 8 characters.
This password is too common.
This password is entirely numeric.
Bypass password validation and create user anyway? [y/N]: y
Superuser created successfully.
```

Migrate database

With the token, we run the function name createSocket(token), this takes and creates a WebSocket request to the backend server. Before running this part make sure to makemigrations and migrate the database.

```
python manage.py makemigrations
python manage.py migrate
```

Run the server in the localhost with port 8000 and create a socket connection with a token parameter in the query params of the request.

```
python manage.py runserver
```

path for WS connection,

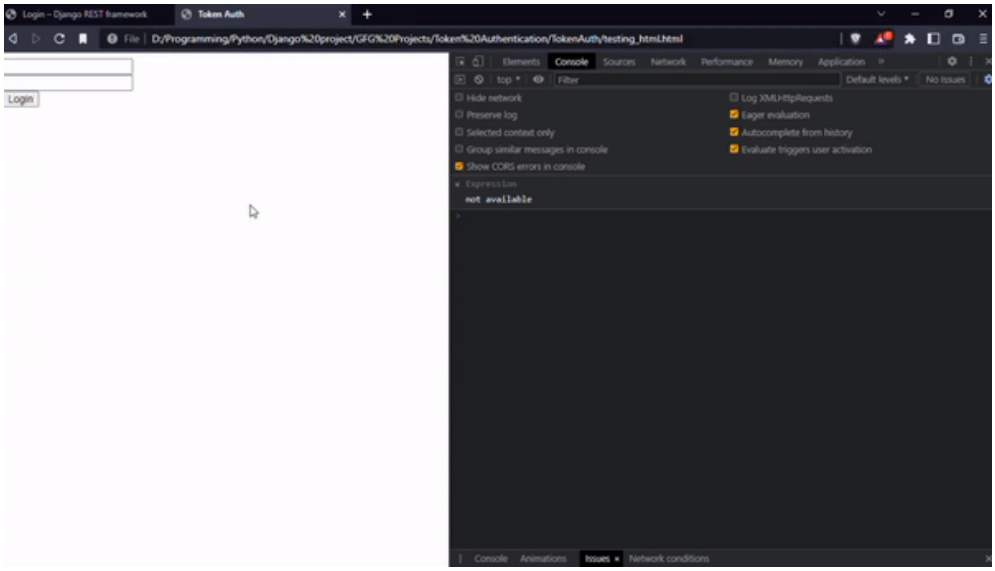
```
ws_path = "ws://localhost:8000/?token="+token ;
```

The token is passed from the success parameter of the ajax post request. Then we create WS connection, once you get connected you will receive the data which was sent by the consumer.

Output:

First, we entered the correct login and password, which caused the token id to appear; second, we entered the incorrect password, which prevented the token from appearing.

To open the HTML file just copy the path of the file and paste it into your browser, or go to the folder and double-click on the file to launch the HTML file.



Are you ready to elevate your web development skills from foundational knowledge to advanced expertise? Explore our Mastering Django Framework - Beginner to Advanced Course on GeeksforGeeks, designed for aspiring developers and experienced programmers. This comprehensive course covers everything you need to know about Django, from the basics to advanced features. Gain practical experience through **hands-on projects** and real-world applications, mastering essential Django principles and techniques. Whether you're just starting or looking to refine your skills, this course will empower you to build sophisticated web applications efficiently. Ready to enhance your web development journey? Enroll now and unlock your potential with Django!

2

Next Article

Django Channels - Introduction and Basic Setup

Similar Reads

Implement Token Authentication using Django REST Framework

Token authentication refers to exchanging username and password for a token that will be used in all subsequent requests so to identify the user on the server...

2 min read

Learn to use Websockets with Django

Django Channels is an extension of the Django framework that allows for handling WebSockets, HTTP2, and other protocols. It integrates with Django's...

4 min read

Django Channels - Introduction and Basic Setup

Django is a powerful Python framework for web development. It is fast, secure, and reliable. Channels allow Django projects to handle HTTP along with...

6 min read

CSRF token in Django

Django provides a feature known as a CSRF token to get away from CSRF attacks that can be very dangerous. when the session of the user starts on a website, a...

6 min read

Simple chat Application using Websockets with FastAPI

In this article, we'll delve into the concept of WebSockets and their role in facilitating bidirectional communication between clients and servers....

6 min read

Python Django | Google authentication and Fetching mails from scratch

Google Authentication and Fetching mails from scratch mean without using any module which has already set up this authentication process. We'll be using...

12 min read

Build an Authentication System Using Django, React and Tailwind

In this article, we will guide you in building an Authentication system using React and Tailwind with the Django Framework that includes features like sign up, log...

15+ min read

JWT Authentication with Django REST Framework

JSON Web Token is an open standard for securely transferring data within parties using a JSON object. JWT is used for stateless authentication mechanisms for...

2 min read

Securing Django Admin login with OTP (2 Factor Authentication)

Multi factor authentication is one of the most basic principle when adding security for our applications. In this tutorial, we will be adding multi factor authentication...

2 min read

Django Authentication Project with Firebase

Django is a Python-based web framework that allows you to quickly create efficient web applications.. When we are building any website, we will need a s...

7 min read

Article Tags : Django Python Technical Scripter Python Django +1 More

Practice Tags : python

Corporate & Communications Address:-
A-143, 9th Floor, Sovereign Corporate
Tower, Sector- 136, Noida, Uttar Pradesh
(201305) | Registered Address:- K 061,
Tower K, Gulshan Vivante Apartment,
Sector 137, Noida, Gautam Buddh
Nagar, Uttar Pradesh, 201305