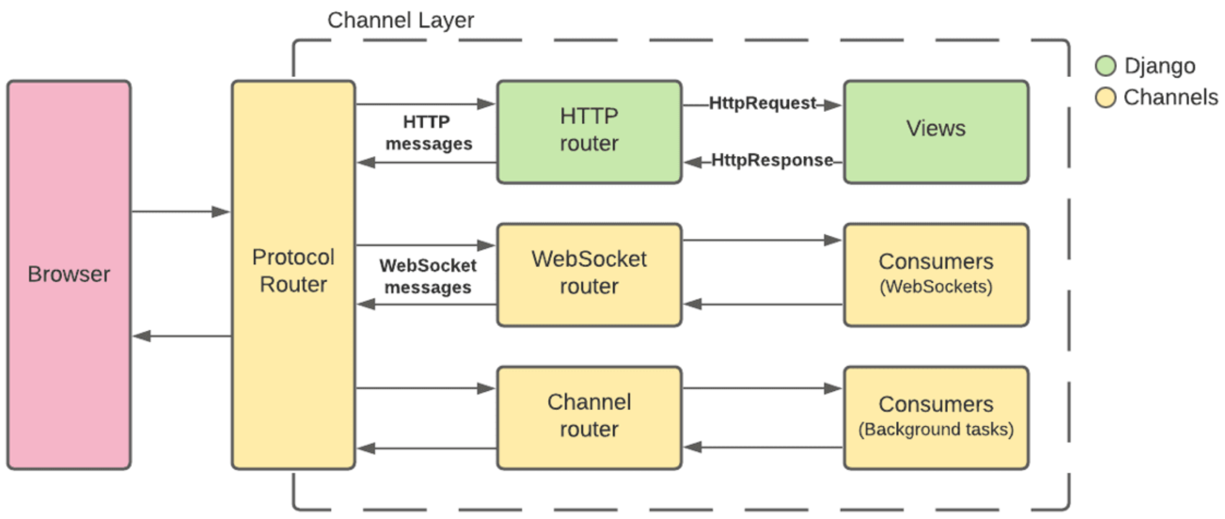
Django channels and WebSocket

First let’s talk about need of Channels and WebSocket. Like we already have normal HTTP request so why can't we use normal HTTP request?

HTTP request: In the normal HTTP request first, the connection request is sent by client to server then the acknowledge signal is provided. It is like a proper handshaking method by which connection is established. So, when the request is fulfilled by the server then at that point of time the connection is terminated. So, if we want the channel not to be terminated again and again then normal HTTP will not work here.



Channels: Channels is a project that takes Django and extends its abilities beyond HTTP - to handle WebSocket, chat protocols, IoT protocols, and more. It's built on a Python specification called ASGI. When a connection request is sent by client to server then the acknowledge signal is provided. It is like a proper handshaking method by which connection is established. So, when the request is fulfilled by the server then at that point of time the connection is not terminated. The connection remains persistent until termination is initiated by either client or any server failure.

So, this is the main reason that we use Django channels with WebSocket.

Channels built upon the native ASGI support in Django. Whilst Django still handles traditional HTTP, Channels gives you the choice to handle other connections in either a synchronous or asynchronous style.

❖ Getting started with Django channels:

So, if you are working with Django=4.0.0 then you have to run this command:

“pip install channels”

If you are working with above version of Django=4.0.0 then you have to run the following command:

“pip install -U channels["daphne"]”

Create your virtual environment and create a folder for your Django project.

create Django project:

“django-admin startproject tictac”

Now open your settings.py file and first add “channels” in Apps

“channels”

Then you will find an object who is referring to the WSGI file. The object is located under templates directory:

```
“WSGI_APPLICATION = 'tictac.wsgi.application'”
```

Just comment this line and create a new object for ASGI.

```
ASGI_APPLICATION = 'tictac.asgi.application'
```

Here tictac is the name of my project so just replace it with your project name.

So before creating the ASGI file let’s understand something about ASGI.

ASGI, or the Asynchronous Server Gateway Interface, is the specification which Channels and Daphne are built upon, designed to untie Channels apps from a specific application server and provide a common way to write application and middleware code.

It’s a spiritual successor to WSGI, designed not only run in an asynchronous fashion via asyncio, but also supporting multiple protocols.

The full ASGI spec can be found at https://asgi.readthedocs.io

Now add this in “settings.py”

```
CHANNEL_LAYERS = {
    "default": {
        "BACKEND": "channels_redis.core.RedisChannelLayer",
        "CONFIG": {
            "hosts": [("localhost", 6379)],
        },
    },
}
```

Before that make sure that Redis is working properly in your system. You can replace Redis with RabbitMQ.

Here all work is done in settings.py file. Now let’s create an ASGI file.

So, you have to work in ASGI file in your main directory in which “settings.py”

File is there.

Copy and paste the content:

```
import os
from channels.routing import ProtocolTypeRouter,URLRouter
from channels.auth import AuthMiddlewareStack
```

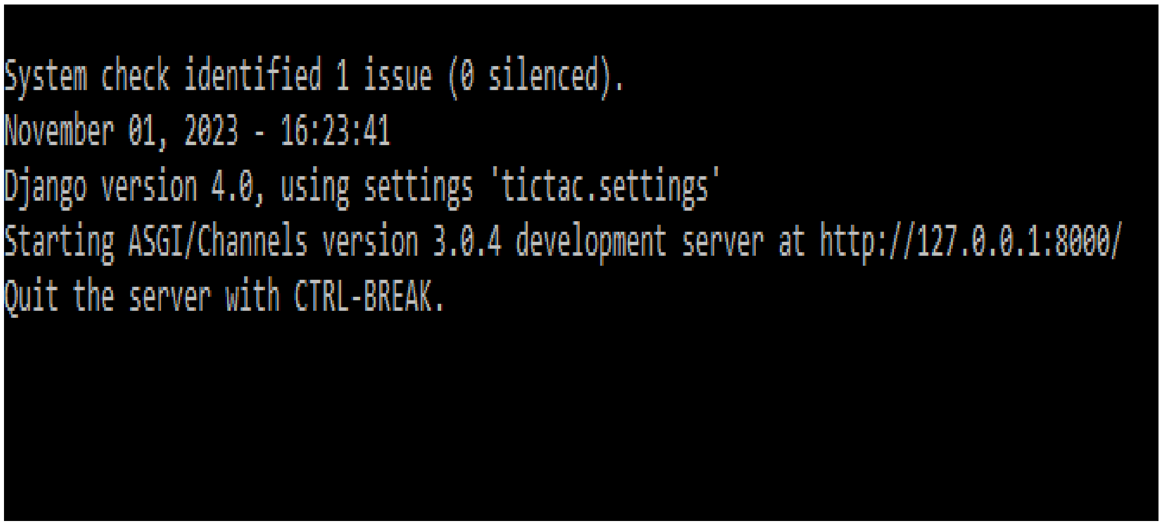
```python
from home.consumers import GameRoom
from django.core.asgi import get_asgi_application
from django.urls import path
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'tictac.settings')


application = get_asgi_application()
```

Make sure to replace “tictac” with your original project name.

Now save the file and run the project:

“python manage.py runserver”

You will see output like this:



```
System check identified 1 issue (0 silenced).
November 01, 2023 - 16:23:41
Django version 4.0, using settings 'tictac.settings'
Starting ASGI/Channels version 3.0.4 development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

Now ASGI/Channels development server will start.

Now let’s create URL for our WebSocket as normally we create URL in “urls.py” file but here we will define our URLS in ASGI file itself.

```
ws_pattern = [
        path('ws/game/<room_code>' , GameRoom.as_asgi())
]



application= ProtocolTypeRouter(
    {
        'websocket':AuthMiddlewareStack(URLRouter(
            ws_pattern
        ))
    }
)
```

Here I am creating a room for a game you can define your path as per your need.

Now as normally when we create URLs then we have to define views for that so same as in Channels when we define URLs then we need to define consumers for URLs patterns.

A consumer is the basic unit of Channels code. We call it a *consumer* as it *consumes events*, but you can think of it as its own tiny little application. When a request or new socket comes in, Channels will follow its routing table - we’ll look at that in a bit - find the right consumer for that incoming connection and start up a copy of it.

This means that, unlike Django views, consumers are long running. They can also be short-running - after all, HTTP requests can also be served by consumers - but they’re built around the idea of living for a little while (they live for the duration of a *scope*, as we described above).

So, create an app:

“python manage.py startapp home”

Now add your app in settings.py and migrate the project

After that create a new file in your app and name it “consumers.py”

Paste the code:

```
from channels.generic.websocket import WebsocketConsumer

from asgiref.sync import async_to_sync

import json




class GameRoom(WebsocketConsumer):

    def connect(self):

        print("fun called")

        self.room_name = self.scope['url_route']['kwargs']['room_code']

        self.room_group_name = 'room_%s' %  self.room_name

        print(self.room_group_name)



        async_to_sync(self.channel_layer.group_add)(

            self.room_group_name,

            self.channel_name

        )


        self.accept()

        print("conectionn accepted")
    def disconnect(self,*args,**kwargs):
        async_to_sync(self.channel_layer.group_discard)(
            self.room_group_name,
            self.channel_name
        )

    def receive(self , text_data):
        print(text_data)
        async_to_sync(self.channel_layer.group_send)(
            self.room_group_name,{
                'type' : 'run_game',
                'payload' : text_data
            }
        )


    def run_game(self , event):
        data = event['payload']
        data = json.loads(data)


        self.send(text_data= json.dumps({
            'payload' : data['data']
        }))
```

Let’s understand the code:

- First, I used Websocketconsmer. This wraps the verbose plain-ASGI message sending and receiving into handling that just deals with text and binary frames. In simple terms if you want to receive and send text or JSON data using WebSocket then WebSocket consumer is used. There are several types of consumers you can check from here: Click here
- Then I have used "async_to_sync"
- Django Channels uses the ASGI (Asynchronous Server Gateway Interface) protocol to handle asynchronous tasks, such as handling WebSocket connections. However, many Django applications and libraries are built around the traditional synchronous request-response cycle. To bridge this gap, **async_to_sync** is used to adapt synchronous code to work within an asynchronous environment.
- Websocketconsumer provides three types of functions like connect, receive and disconnect. Connect is used for making connection with WebSocket and receive is triggered when data is sent from front-end to WebSocket. Disconnect is used for terminating the connection.
- In the connect method we have to define the room name and room_group_name

```
async_to_sync(self.channel_layer.group_add)(
            self.room_group_name,
            self.channel_name
        )

        self.accept()
```

In this code we have added the group name and channel name so that a group is created. Then we have to accept the connection.

In the receive method

```
async_to_sync(self.channel_layer.group_send)(
            self.room_group_name,{
                'type' : 'run_game',
                'payload' : text_data
            }
        )
```

We are calling a function that is named “run game”. The Run game will send the data to front-end again.

Whole flow:

- Connection is created by connect method
- We can send data from the front-end using “socket.send” in JS.
- Now in backend we can view and manipulate the data using receive method in consumer as we have done.
- We created the “run game” function and this function is sending the data to front-end again.
- We can receive this data in front-end by using “socket.onmessage” and we can do anything we want.

Channels documentation:
https://channels.readthedocs.io/en/latest/introduction.html#

If you are facing issue like ASGI server is not starting, then read this document properly: Click here