Django Channels

What is Django Channels?

Django Channels is a Django project that takes Django beyond the regular HTTP and HTTPS protocols and allows us to use WebSocket, IoT protocols and chat protocols, and more. Django Channels is built on a python specification called ASGI (Asynchronous Server Gateway Interface).

What we want to achieve by using this technique is to create a two-way communication between a web socket and our server, rather than the usual request/response communication structure given by the HTTP and HTTPS protocols. This means, of course, that we will be using another protocol called WS (Web Socket). Parallel to the secure version of the HTTP (Hyper Text Transfer Protocol), the HTTPS (Hyper Text Transfer Protocol Secure), the WS protocol, has also a secure version, the WSS (Web Socket Secure).

Channels and Web Sockets

Whenever we want to create a two-way connection using Django channels we will need two things. One is of course Django channels, and the other one will be a web socket. The Django channels will be on our side, on the server side; and the web socket will be on the client side.

And how will the client know to create a web socket?

Well, inside our HTML there will be a script, written in javascript which will create an object that will inherit from the WebSocket class. This class comes from javascript, not from python. This means that the one who creates the Web socket will be the client himself, because by entering the webpage and sending a message, the WebSocket object will be created, and just like we did with our HTTP, there will be a designated URL for our javascript WebSocket object to connect. On the other end of the URL, there will be a routing waiting. The routing behaves just like the URL list we link to our views, the one difference here, is that we will not link them to a view, we will link it to a different type of function called a consumer.

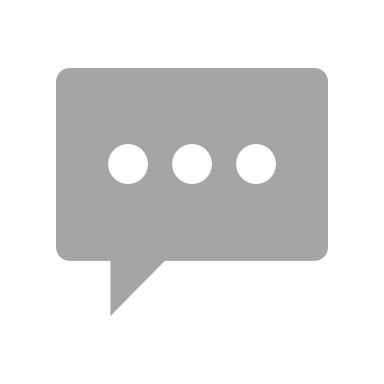
About Sockets

Remember that in the last instance, every programming language compiles its code to the same structure, for the same type of processor. This means, that if python’s print function is to print “Hello World” on the screen, and JavaScript’s console.log will output the same string, in the same place (the terminal), their compiled machine code, for the same processor, has to be the same.

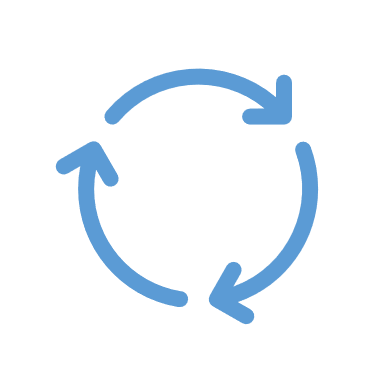
This is how a web socket written on javascript can connect a server or another socket or routing(this I am not clear, but that is the idea)

Difference between HTTP/HTTPS and the WS/WSS protocols

HTTP/HTTPS



**POST**



**Query Db**

John

**on post &**

**every second**

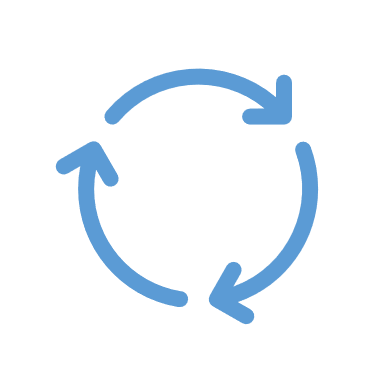
Server

Db



**1s**

**GET**



**Query Db**

Mary

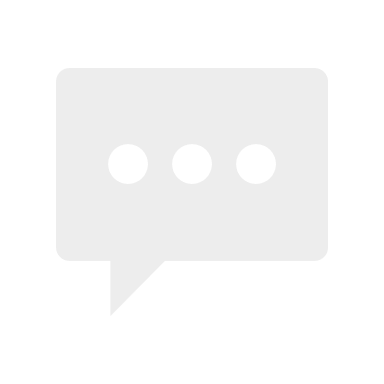
**every second**

In this kind of communication structure, John posts a message to the server, which is queried and saved to the database. Then Mary, to get the message, makes a get request every 1s to the server which queries the database and sends a response back to Mary. Whether there is a new message or not, a response will be given to Mary. In this structure, the same will be happening with John. This means that our server will be querying the database every second, by every user who is using the service. This is of course a huge load for the server.

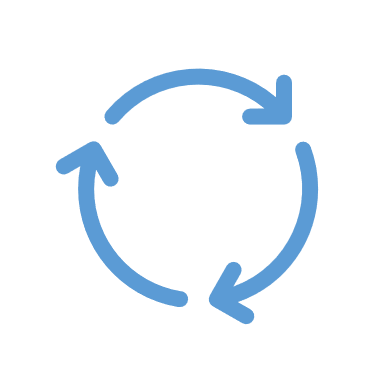
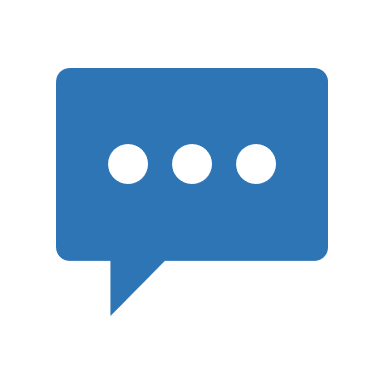
IMPORTANT

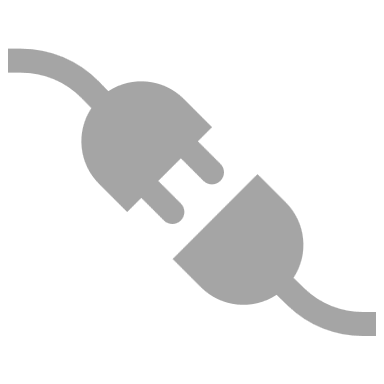
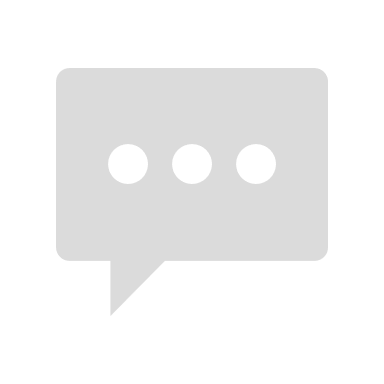
Consider that this graphic was made considering the very moment when john posts a message, and Mary is requesting the page every second, but when John’s POST request is done, he will start the GET request cycle, every second.

WS/WSS



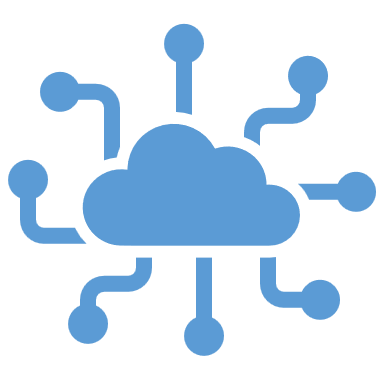
John





**Query Db**

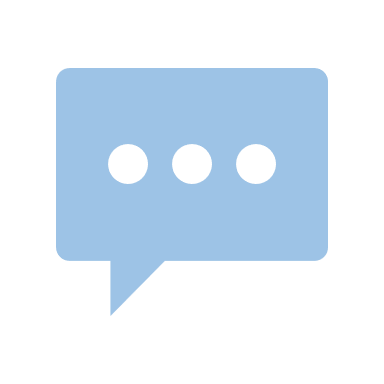
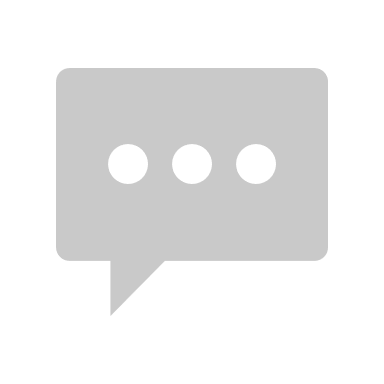
**Web Socket**



**Broadcast**

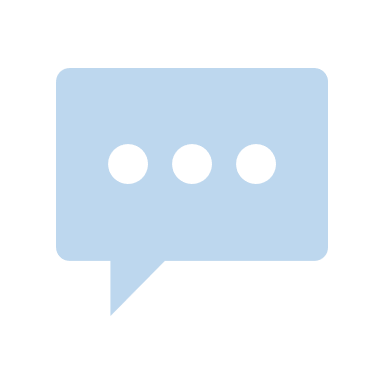
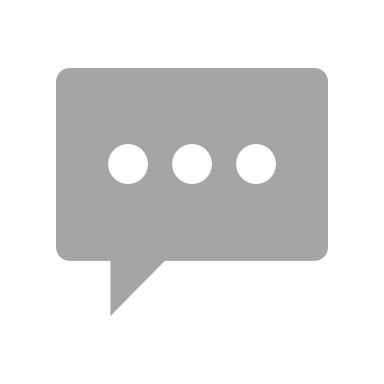
Db

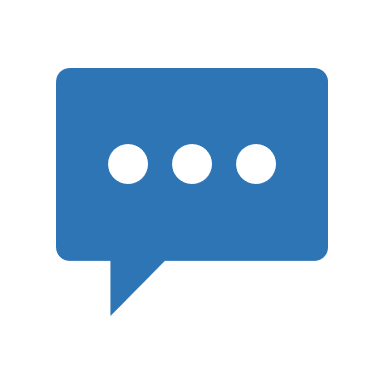
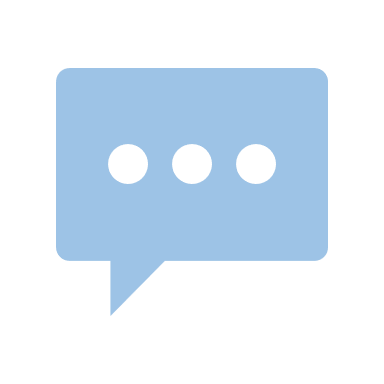
Server



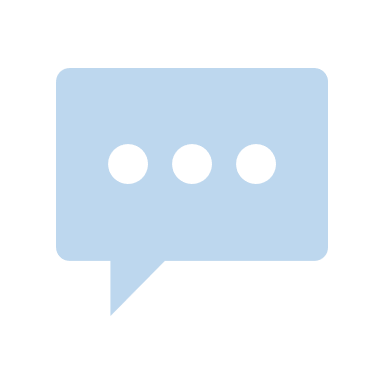
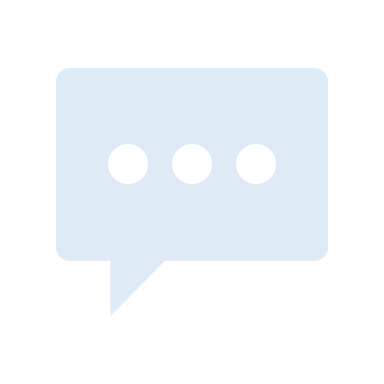
**once**

**on message**

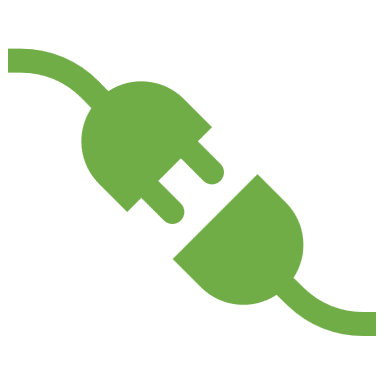
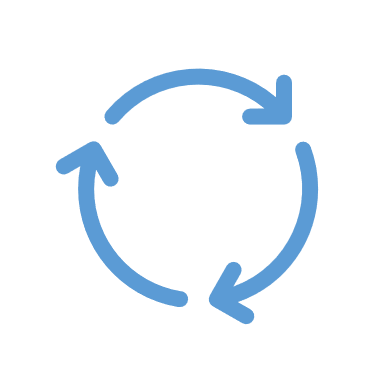
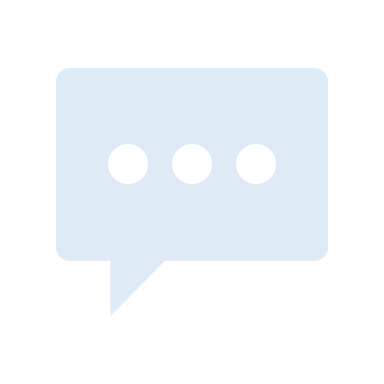




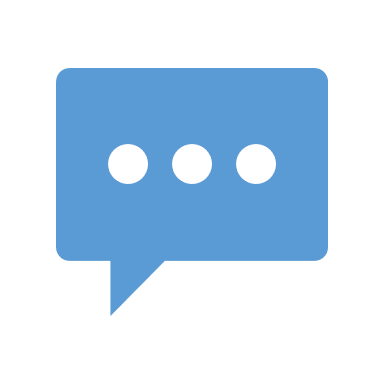
Mary

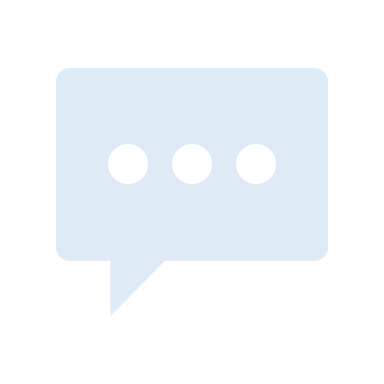
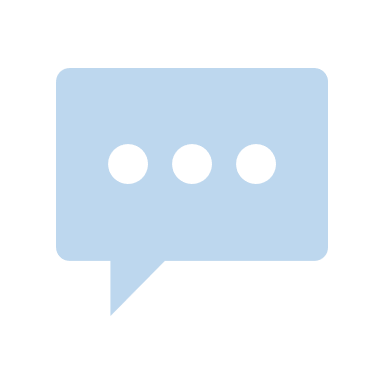


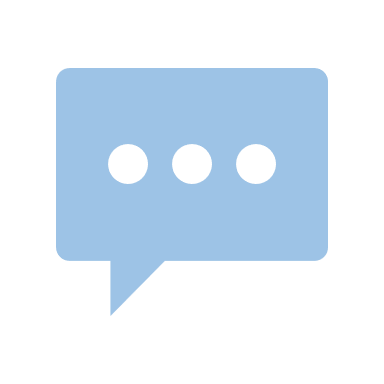
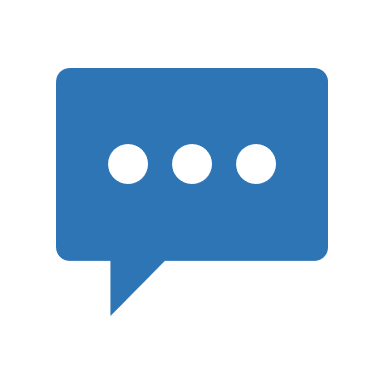
**on message**



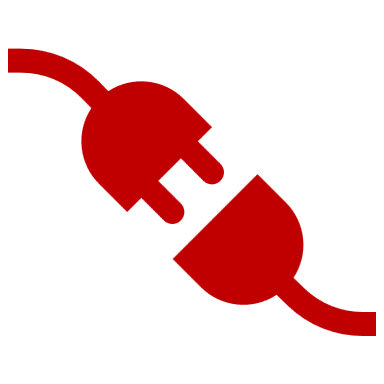
**Web Socket**







Chris



**Web Socket**

The WS protocol is more practical when it comes to sending messages to more than one person, or whenever we need to maintain a connection open between us, and the other person. In this case, John sends a message, but John’s page is not loaded after he sends it, this already means we are not requesting our server to send us a new HTML file with all the information, data, and load that comes with it. Here we are simply passing the information via our WebSocket, which will be connecting to a specific URL that we will allow. All the other members of this chat will be connected to the same room(this concept will be explained later). Whenever John sends a message, this passes to the JavaScript web socket, which tries to connect with our server on the URL. On that URL, we will have a routing waiting for this connection, this router will then, assure that whenever someone goes to that URL, the client will connect to a consumer. Familiar right? We can think of it as the URLs and the views that are waiting on our server.

Once the consumer has the message, this consumer object will know which client is connected to which room, and then he will only send this data via ws, to all the WebSocket objects that are included in that room. All this is done without ever reloading the HTML.

Why is the HTML never reloaded?

The answer is of course on the WebSocket object which lives on the client side (Browser). This web socket is making receiving data in a JSON format. This object has methods like the one we use, the onmessage method. Whenever the WebSocket receives a message, this will fire our script and it will add a new <div> below the last <div>, with our new message. All this can be made without reloading the page, the same way we can do animations, and the page is never reloaded, we can run a script, and the page doesn’t need to reload necessarily.

Installing Channels

First things first, we need to install channels in our virtual environment. This is pretty easy. Just take into account that is user activate because I am using bash on windows, the location of your activate script may vary depending on your setup.

>>>source <your enviroment>/Scripts/activate

>>>pip install channels

The next thing we have to do is to add channels to our installed apps inside our settings.py:

*§settings.py*

# Application definition

INSTALLED\_APPS = [

    'channels',

    'django.contrib.admin',

    'django.contrib.auth',

    'django.contrib.contenttypes',

    'django.contrib.sessions',

    'django.contrib.messages',

    'django.contrib.staticfiles',

    'chat',

]

Configuring ASGI

Now, let’s add our ASGI application to our settings.py. This application is a variable that lives inside asgi.py which we will import the same way we import our WSGI application from the wsgi.py file.

*§settings.py*

ASGI\_APPLICATION = 'AppsRepo.asgi.application'

Now, we must make some adjustments to our application variable to support both HTTP and WS protocols.

*§asgi.py*

1 import os

2

3 from django.core.asgi import get\_asgi\_application

4 from channels.routing import ProtocolTypeRouter, URLRouter

5 from channels.auth import AuthMiddlewareStack

6 import automation.routing

7 os.environ.setdefault('DJANGO\_SETTINGS\_MODULE', 'AppsRepo.settings')

8

9 application = ProtocolTypeRouter({

10     'http': get\_asgi\_application(),

11    'websocket': AuthMiddlewareStack(

12        URLRouter(

13            automation.routing.websocket\_urlpatterns

14        )

15    )

16 })

This might look like a lot but is simple. The only thing we are changing here is the variable application. This variable is by default set to get\_asgi\_application(). What we are doing, is simply specifying that whenever we use HTTP, yes, use the default configuration. But, whenever we use the websocket protocol, our application will be the AuthMiddlewareStack(). Everything we are importing from lines 4 to 6 objects we are going to use to set our application whenever we use the WebSocket protocol. We can see that in line 13, we are using a variable called websocket\_urlpatterns, this variable is passed to the URLRouter object, which is passed to the AuthMiddlewareStack object.

Client’s Web Socket object

*<>chat.html*

1 <script type="text/javascript" >1

2

3     //End Point to start the handshake "windows.location.host" is our root, and

4 "/ws/socket-server" is just a url we will use for establishing the connection

5     let url = `wss://${window.location.host}/ws/socket-server`

6

7     //This is the instance of the Websocket object

8     const chatSocket = new WebSocket(url)

9

10    chatSocket.onmessage = function(e){

11       let data = JSON.parse(e.data)

12       console.log("Data :",data)

13 }

14 </script>

We have created then, on lines 10-13 a function that uses the WebSocket objects onmessage method, which will fire this function. We will see in our consumer ( the other end ) a special method for the connection, we will make our web socket send a message saying: *“connection established”*, whenever we go to our routing URL to test the connection. At this point, if we try to connect, we will get an error because there is nothing on that URL on our server. So the next step, once we have our Web Socket from the client, is to create an endpoint on that URL, a consumer which will handle the ws connections. We will create both the consumers and the routings so we can establish a connection and then we will continue to the client side.

Consumers

The consumers.py file will be created in the same place as all the other scripts. Consumers are like the views from Django, except that they can also initiate communication and send data to the client’s web socket, whenever they see fit. Consumers will inherit from the class WebsocketConsumer. As its name indicates, this object is the one that connects to the WebSocket object we created before on the javascript part of the HTML.

Now we will see the minimal configuration for testing our consumer’s connection to the web socket by simply connecting, accepting, and then sending data using a JSON object. The connect method will allow the connection between these two objects, the accept method will accept the connection. We will modify these two methods in the future so we can connect clients to the right consumer, and then specify better with the channel layers.

*§consumers.py*

import json

from channels.generic.websocket import WebsocketConsumer

class ChatConsumer(WebsocketConsumer):

    def connect(self):

        self.accept()

        #Here the consumer sends a message our Web Socket

        self.send(text\_data = json.dumps({

            'type': 'connection\_established',

            'message': 'You are connected,

        }))

We can see that the connect function has two methods that have been inherited from the mother class, the methods self.accept and the method self.send. In the accept method we were forced to include them because if we don’t, nothing will happen. The send method, on the other hand, was included as a test. What will happen here is that whenever a Web Socket connects to this WebsocketConsumer, this will send a message, then, on the front end we have established a [function](#Socket_onmessage) which, upon receiving a message, will console.log the message on the console.

Routers

This is a rather simple configuration we do the same as we would do in our urls.py file. If our URLs fire our view, and they are accessed via HTTP, then our WebSocket urlpatterns will fire our consumer, but it will fire the as\_asgi() method. You can think of this method as the as\_view() method we would use normally when rendering a view using a class, because ChatConsumer is not a function, but a class.

The path we will connect to this WebsocketConsumer object will be of course we set our client’s WebSocket object to go to

*§routing.py*

from django.urls import re\_path

from . import consumers

websocket\_urlpatterns = [

    re\_path(r'ws/socket-server', consumers.ChatConsummer.as\_asgi())

]

Sending and Receiving Messages

Now that we have the structure settled and we can both send and receive messages, let’s actually set a from where we can send data from the client’s WebSocket to our WebsocketConsumer, and also set a receive method inside our WebsocketConsumer, which will handle the data received, save it to the database ( optional ), and then we will want to broadcast this message. We will do so by using the already-known [send](#WebsocketConsumer_send) method from the WebsocketConsumer class.

*<>chat.html*

1 <h1>ChatApp</h1>

2 <form action="" id="form">

3 <input type="text" name="message">

4 </form>

5 <script type="text/javascript" >1

6

7     let url = `ws://${window.location.host}/ws/socket-server`

8     const chatSocket = new WebSocket(url)

9

10    chatSocket.onmessage = function(e){

11       let data = JSON.parse(e.data)

12       console.log("Data :",data)

13 }

14 //Now, instead of sending our data using our form as we normally would, we will prevent

15 //that from happening and then we will use that data and send it to our WebSocket,

16

17 //getting the form, we will work with

18 let form = document.getElementById("message\_form")

19

20 //EventListener to prevent the form from being submitted

21 form.addEventListener('submit', (e)=> {

22     e.preventDefault()

23     let message = e.target.message.value

24     //Sending the data using our Websocket and JSON

25     chatSocket.send(JSON.stringify({

26        'message':message,

27        'from\_user':'{{user}}',

28        'to\_user':'{{user2}}'

29     }))

30 //Setting the form to blank

31 form.reset()

32 })

33 </script>

Explanation:

First, we will create a way of taking the data and passing it to our WebSocket and we will do this by simply creating a [form](#form). This form must have at least an input inside so we can pass at least a text string as a message. We do this on lines 2-4.

Then we need to get the data from this form, convert it to JSON format and then transmit it to our WebsocketConsumer. To do this, we need to [prevent](#preventdefault) the form from being submitted and generate a POST request which would cause the page to load. We do this in lines 20 (Listen for submission) and line 21 (Prevent the form from being submitted and generate a post request). Next, we need to get the data which would normally be passed to the backend as a POST request. This data is the data being held by the input with the id message, which is inside the form. This is why we can access it via e.target.message.value because the e.target is resolved to the form object, then inside the form object, we access the object with the name [message](#message_input). There is no need for this, we could normally use a document.getElementById and then get the value, and it would have the same effect.

Then the only thing we need to do is to pass all this data by using the [send](#Websocket_send) method from the WebSocket object and passing a JSON object string as data. Also, as a mental note, we can still use the variables provided by Django using the Django tags.

Finally, we [reset](#form_reset) the form so it looks like the page was loaded.

Channel Layers

Before jumping to the consumers, we have to set up channel layers. Channel layers allow consumers to talk to each other, among other things. Let’s remember that whenever we are using Django channels, we usually establish a connection between one WebsocketConsumer and several web sockets. With channel layers, we allow the WebsocketConsumer objects to talk to each other.

A channel layer provides us with two main features: groups and channels. Groups are basically like chat rooms that store information about users in a particular chat room, and this room will be stored in an In-Memory Database. Inside these groups or rooms, we have channels. The channels are representing the users inside that specific group, and if you know someone’s specific channel name you can talk to that person.

Whenever we create a room we will have a group representing that room, and inside of that group, we will have channels representing the users. This means that if a user wants to start receiving messages from that chat room ( group ), the user’s channel needs to be added to that group (see [figure](#websocket_broadcast) ).

To enable channel layers we just have to go to our settings.py and add the variable CHANNEL\_LAYERS with a dictionary, inside another dictionary. Here we will use the InMemoryChannelLayer object and set it as default for the BACKEND key.

ASGI\_APPLICATION = 'AppsRepo.asgi.application'

CHANNEL\_LAYERS = {

    'default':{

        "BACKEND": 'channels.layers.InMemoryChannelLayer'

    }

}

We will cover the right configuration for channel layers using Redis later, for now, let’s continue to store the data referring to what layer belongs to what user using this object.

Uvicorn

In order to use the WS and WSS in a production environment, we need something more than just gunicorn, since gunicorn cannot handle ASGI applications or WebSockets. To solve this, we have to install uvicorn, which will act as an ASGI webserver. More precisely, we will use uvicorn’s workers and then make them work with our gunicorn server.

Installation command:

python -m pip install uvicorn[standard]

Note: It is important to use the standard version because it can handle some WebSocket libraries that the nonstandard one can’t.

Then, we can use this command, which is the same as always, but with the difference that now we specify that we want gunicorn to work with the uvicorn’s workers.

gunicorn -c gunicorn\_config/gunicorn\_config.py AppsRepo.asgi -k uvicorn.workers.UvicornWorker