



Step by Step Django Channels

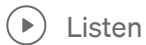
Build Real-Time Apps with Django Channels



Okan Yenigün · [Follow](#)

Published in [Level Up Coding](#)

11 min read · Nov 29, 2022



Listen



Share



Photo by [Volodymyr Hryshchenko](#) on [Unsplash](#)

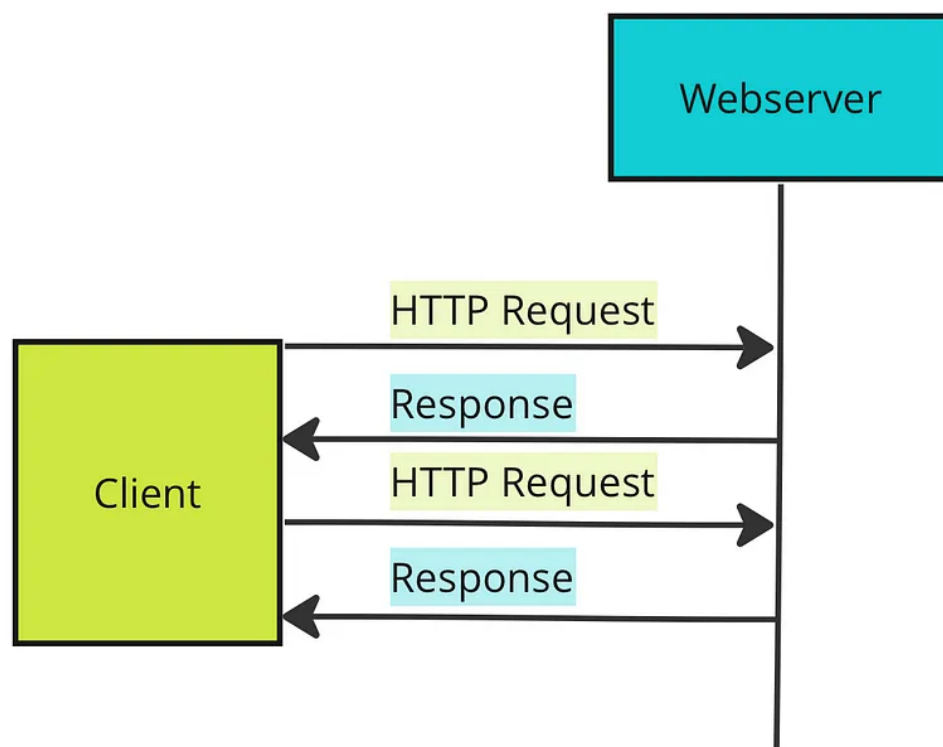
Django normally works with HTTP responses in a synchronous fashion. We use Django Channels to develop real-time, dynamically updated applications that can work

asynchronously by extending Django's HTTP abilities. For example, you want to develop a chat application or an application that processes real-time data from an IoT sensor, or an application that can work with a WebSocket for any purpose, with Django. In such a case, the extension you should refer to is Django Channels.

Synchronous vs Asynchronous

Let's first look at what synchronous and asynchronous work is in terms of Django.

Django works synchronously. That is, an HTTP request is handled completely synchronously. A request is sent, waited and the response is returned.



Synchronous. Image by the author.

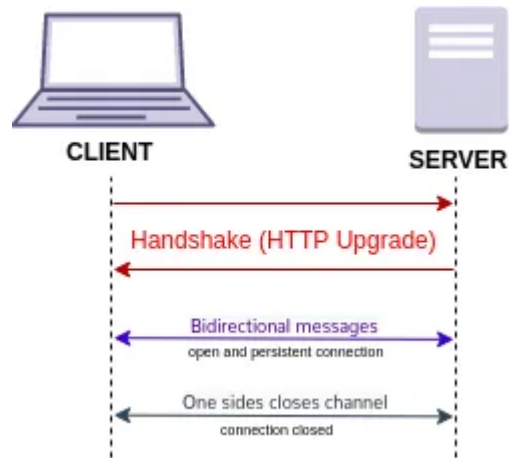
The server is not able to send a single message to the client. In a synchronous operation, there should be a request to get a response.

In asynchronous operation, we launch a request. There is no waiting for a response, the app continues to serve the user and executes the tasks.

Websockets

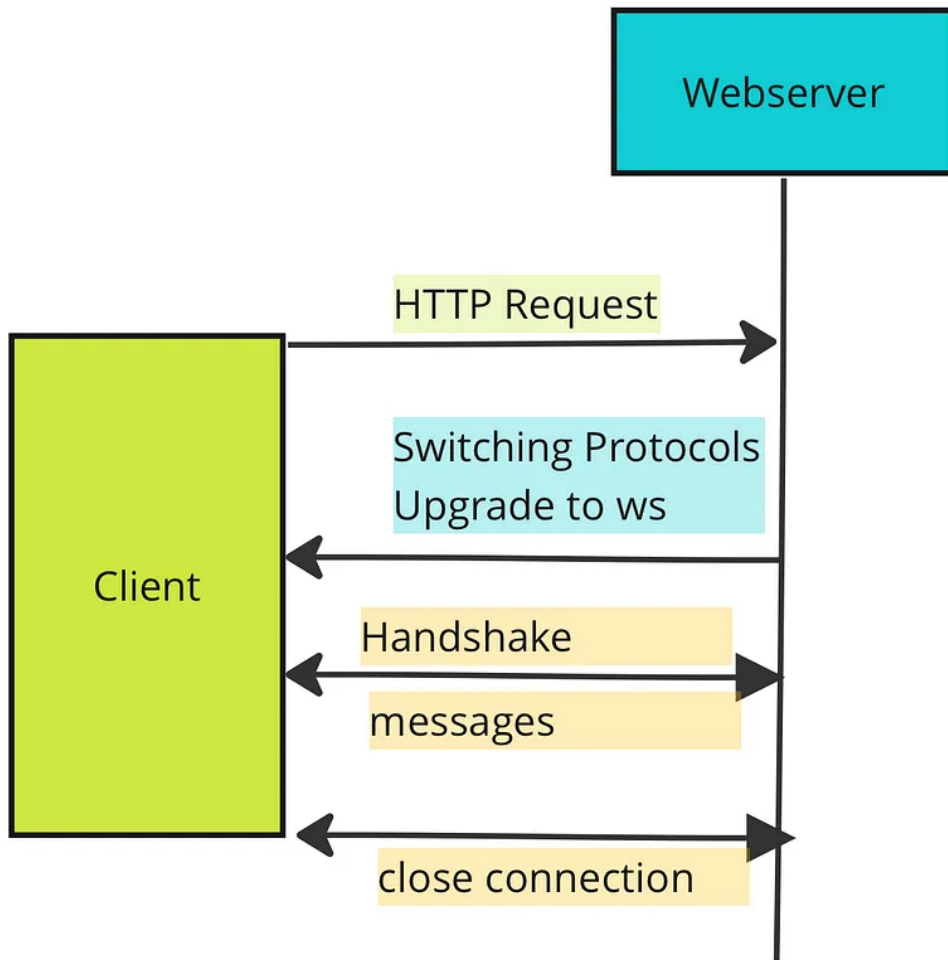
Websockets is a protocol that can establish two-way communication between the browser and the server (HTTP is a one-way protocol). A client can send a message to a

server and receive a message about the relevant event without having to wait for a response. Both parties can communicate with each other independently at the same time.



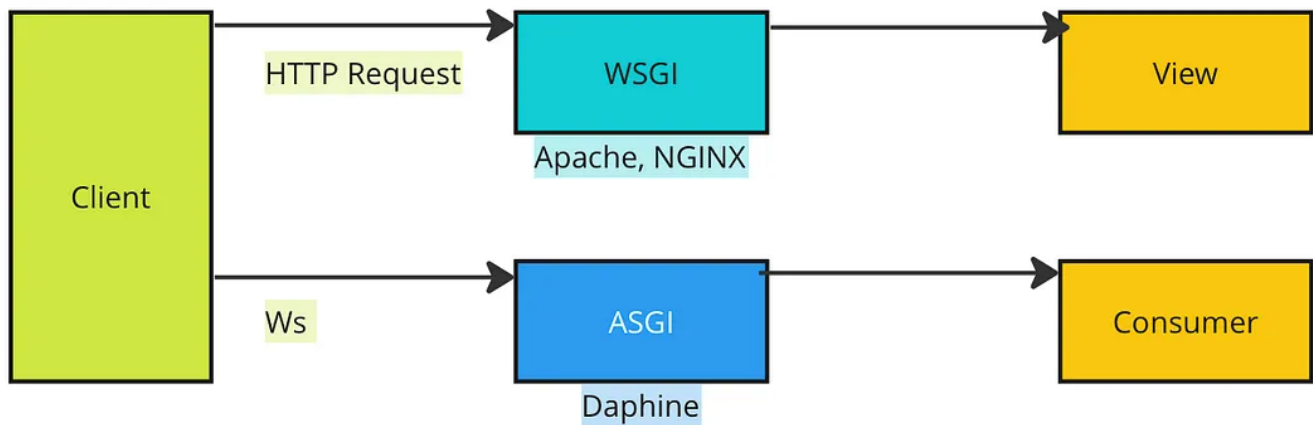
A diagram describing a connection using WebSocket. Source: [Wikipedia](#)

Websocket is a stateful protocol, meaning the connection between client and server stays alive until terminated by one of the parties (client or server). After closing the connection by either client or server, the connection is terminated from both ends.



Websocket Connection. Image by the author.

In the first step, the client sends an HTTP request to the server. It asks the server to open a WebSocket connection. The server accepts it and returns a 101 switching protocols response. At this point, the handshake is completed. TCP/IP connection is left open and both sides can send messages. The connection remains open until one of them drops off. This process is often referred to as full-duplex.



Comparison of HTTP and Websockets in Django App. Image by the author.

When a client sends an HTTP request, it is received by a Django application via WSGI (web server gateway interface). It ends up in Django's URL and is routed to Django's view.

For WebSockets, ASGI (asynchronous server gateway interface) is in charge instead of WSGI. And it is routed to a consumer instead of a view.

You can get the project code [here](#).

[Documentation of channels package.](#)

pip install channels

Example 1

In this example, we will build a real-time system that updates a counter in a *div* HTML element. This app will work only for one user. So, it is not the best use case for channels but a simple start.

Example 1 Page. Image by the author.

The business logic is a random word generator. It will display a real-time generated random word on the page without refreshing the page.

```
#business.py

import json
#pip install random_word
from random_word import RandomWords

class Domain:

    def __init__(self):
        self.R = RandomWords()

    def do(self):
        word = self.R.get_random_word()
        return json.dumps({"message": word})
```

```
#urls.py

from django.urls import path
from one.views import one
```

```
urlpatterns = [  
    path('one/', one),  
]
```

```
#views.py  
from django.shortcuts import render  
  
def one(request):  
    return render(request, './templates/one.html', context={'one_text': "ASD"})
```

We have a simple page to display the passed word.

```
{% include 'base.html' %} {% block content%} {% load static %}  
<div class="container">  
    <p id="one">{{ one_text }}</p>  
</div>  
  
<script>  
    var socket = new WebSocket("ws://localhost:8000/ws/any_url/");  
    socket.onmessage = function (event) {  
        var data = JSON.parse(event.data);  
        console.log(data);  
        document.querySelector("#one").innerText = data.message;  
    };  
</script>  
{% endblock %}
```

Don't forget to add channels (and your apps) into installed apps.

```
#settings.py  
  
INSTALLED_APPS = [  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',
```

```
'django.contrib.sessions',
'django.contrib.messages',
'django.contrib.staticfiles',

'channels',
'one',
]
```

We need to update the asgi file since we are using WebSockets. We set our application variable as a *ProtocolTypeRouter* type. It checks the type of connection and protocol.

If the protocol type is correct, then it is given to the *AuthMiddlewareStack* which checks the authentication of the user. A *URLRouter* that takes our consumer routings as an argument is passed into middleware.

```
#asgi.py

import os

from django.core.asgi import get_asgi_application

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

from one.routing import ws_urlpatterns

os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'examplechannels.settings')

application = ProtocolTypeRouter(
    {
        'http': get_asgi_application(),
        'websocket': AuthMiddlewareStack(URLRouter(ws_urlpatterns)),
    }
)
```

To use the asgi, we must define it in the *settings.py* (just copy and edit the default *WSGI_APPLICATION* variable).


```
#settings.py

WSGI_APPLICATION = 'examplechannels.wsgi.application'

ASGI_APPLICATION = 'examplechannels.asgi.application'
```

Let's create a consumer (~view for WebSockets). For this example, we only need to override the *connect* method of the parent class (*WebsocketConsumer*).

It will simply get data from the domain object a hundred times and each time it will sleep for two seconds.

```
import time
from channels.generic.websocket import WebsocketConsumer
from one.bussiness import Domain

class OneConsumer(WebsocketConsumer):

    def connect(self):
        self.accept()
        D = Domain()
        for i in range(100):
            data = D.do()
            self.send(data)
            time.sleep(2)
```

We need to create routing for the consumers just like we create URLs.

```
#routing.py

from django.urls import path
from one.consumers import OneConsumer

ws_urlpatterns = [
    path('ws/any_url/', OneConsumer.as_asgi())
]
```

Let's start the server by simply putting the command: *python manage.py runserver*

```
System check identified no issues (0 silenced).
November 28, 2022 - 19:17:33
Django version 4.0, using settings 'examplechannels.settings'
Starting ASGI/Channels version 3.0.4 development server at
http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Let's go to the page: *http://127.0.0.1:8000/one/*

```
HTTP GET /one/ 200 [0.03, 127.0.0.1:50603]
HTTP GET /static/css/one-style.css?1669663076 200 [0.01, 127.0.0.1:50603]
WebSocket HANDSHAKING /ws/any_url/ [127.0.0.1:50605]
WebSocket CONNECT /ws/any_url/ [127.0.0.1:50605]
```

Let's check the console in the browser:

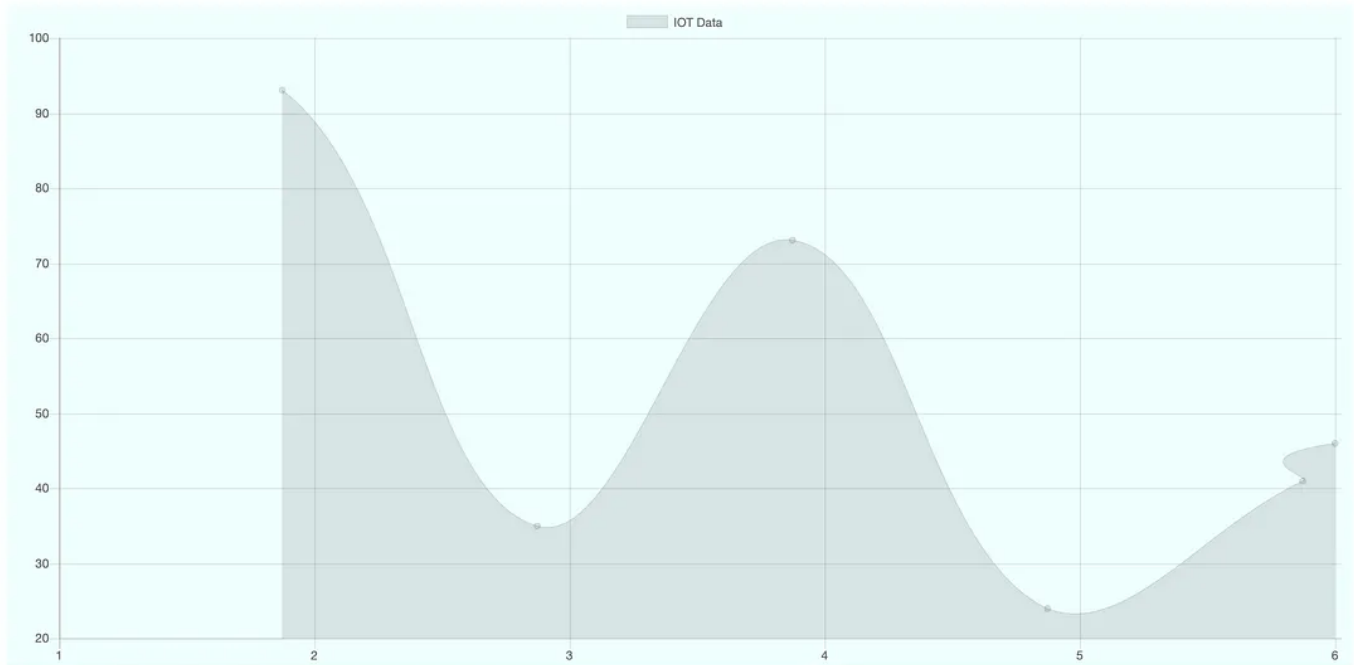


The console of the page of one app. Image by the author.

A new object that carries a word is sent every two seconds.

Example 2

This time let's update a graph in real time as if simulating an IoT mechanism. A new app is created called *two*.



Page of the second app. Image by the author.

We will use Chartjs to display our IoT data. So, we have a *canvas* tag. And, we need to use Chartjs cdn to use it. We have another JavaScript file named *two.js*. It will handle the chart operations.

```
{% include 'base.html' %} {% block content%} {% load static %}
<link rel="stylesheet" href="{% static 'css/one-style.css' %}?{% now 'U' %}" />

<div class="container">
  <div class="chart">
    <canvas id="iot-chart" width="800" height="400"></canvas>
  </div>
</div>
<script src="https://cdnjs.cloudflare.com/ajax/libs/Chart.js/2.4.0/Chart.min.js">
```

```
<script src="{% static 'js/two.js' %}"></script>
{% endblock %}
```

Our domain class simply returns a random integer.

```
#business.py

import json
import random

class DomainTwo:

    def do(self):
        value = random.randint(0,100)
        return json.dumps({'data': value})
```

We have similar routing, URLs, and views:

```
#routing.py

from django.urls import path
from two.consumers import TwoConsumer

ws_urlpatterns = [
    path('ws/two_url/', TwoConsumer.as_asgi())
]
```

```
#urls.py

from django.urls import path
from two.views import two

urlpatterns = [
```

```
    path('two/', two),  
]
```

```
#views.py  
  
from django.shortcuts import render  
  
def two(request):  
    return render(request, './templates/two.html')
```


This time we are constructing an asynchronous working channel. Therefore, we are using *AsyncWebsocketConsumer* and the *async-await* structure.

```
#consumers.py  
  
import time  
from channels.generic.websocket import AsyncWebsocketConsumer  
from asyncio import sleep  
from two.business import DomainTwo  
  
class TwoConsumer(AsyncWebsocketConsumer):  
  
    async def connect(self):  
        await self.accept()  
        D = DomainTwo()  
        for i in range(100):  
            data = D.do()  
            await self.send(data)  
            await sleep(2)
```

To make the app work asynchronously, each client must get a new instance of the consumer. In order to achieve it, we will use **channel layers**. These layers are FIFO (first in first out) data structures. They carry messages received from clients in a queue data structure. We need to install Redis in this case. Redis is an in-memory data store.

For MACOS: *brew install redis*

After installation completes, type *redis-server* in the command prompt:



```
redis-server
32 (it was originally set to 256).
11052:M 28 Nov 2022 23:07:04.930 * monotonic clock: POSIX clock_gettime

Redis 7.0.5 (00000000/0) 64 bit

Running in standalone mode
Port: 6379
PID: 11052

https://redis.io

11052:M 28 Nov 2022 23:07:04.931 # WARNING: The TCP backlog setting of 511 cannot be enforced because kern.ipc.somaxconn is set to the lower value of 128.
11052:M 28 Nov 2022 23:07:04.931 # Server initialized
11052:M 28 Nov 2022 23:07:04.931 * Ready to accept connections
```

Run redis-server. Image by the author.

To be able to use it properly, we also need to install channels-redis:

pip install channels-redis

And we define channel layers in the settings file:

```
#settings.py

CHANNEL_LAYERS = {
    'default' : {
        'BACKEND' : 'channels_redis.core.RedisChannelLayer',
        'CONFIG': {
            'hosts': [('127.0.0.1', 6379)]
        }
    }
}
```

6739 is the port that the Redis server uses.

When we run the server and go to the URL of the second app:

```
System check identified no issues (0 silenced).
November 28, 2022 - 21:07:30
Django version 4.0, using settings 'examplechannels.settings'
Starting ASGI/Channels version 3.0.4 development server at
http://127.0.0.1:8000/
Quit the server with CONTROL-C.
HTTP GET /two/ 200 [0.03, 127.0.0.1:51340]
HTTP GET /static/css/one-style.css?1669669655 200 [0.01, 127.0.0.1:51340]
HTTP GET /static/js/two.js 200 [0.02, 127.0.0.1:51341]
WebSocket HANDSHAKING /ws/two_url/ [127.0.0.1:51343]
WebSocket CONNECT /ws/two_url/ [127.0.0.1:51343]
```

The WebSocket is running. New data comes in every two seconds.

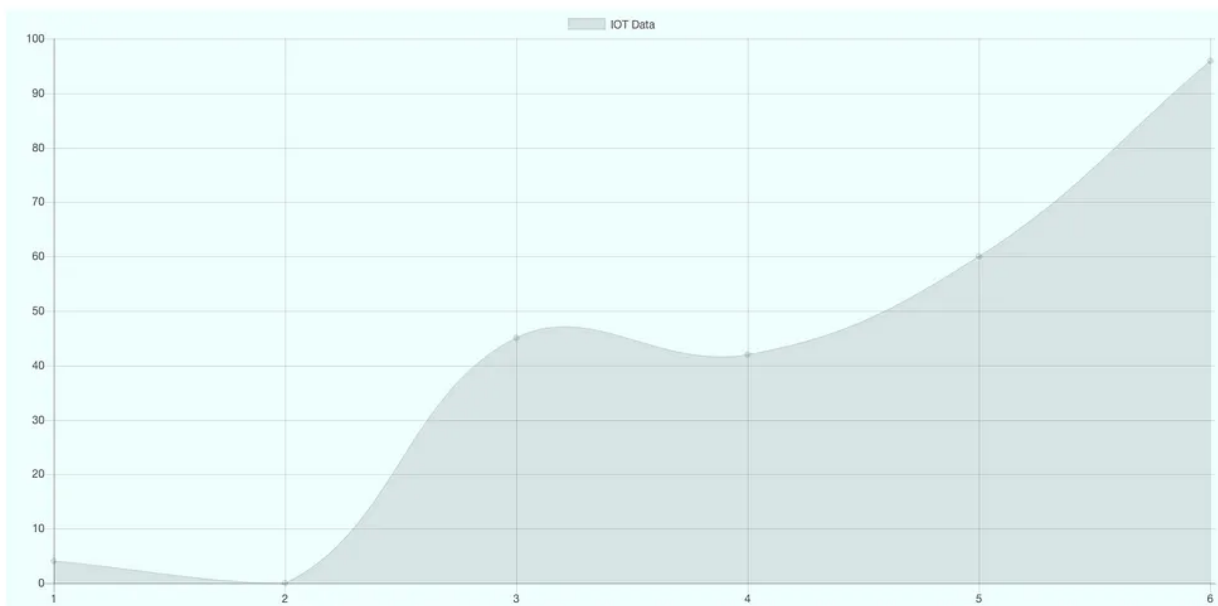
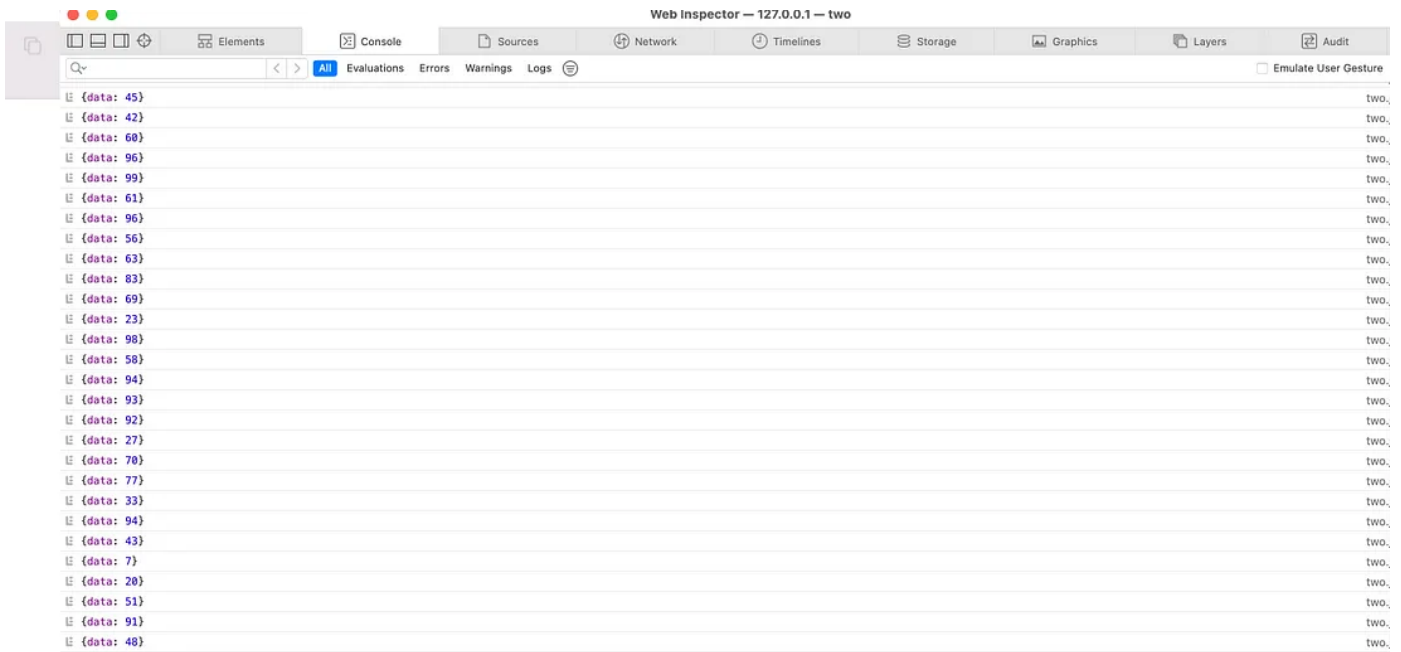


Chart. Image by the author.



Console. Image by the author.

Example 3

This time let's build a chat app.

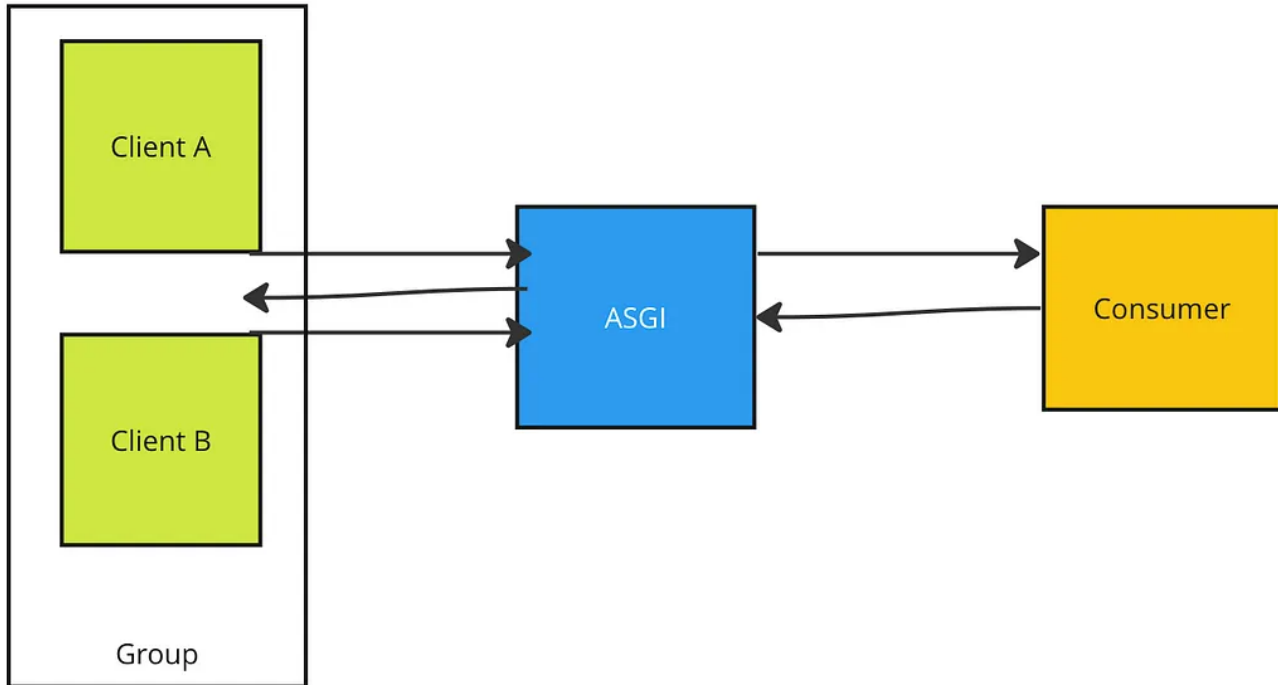
Chatroom - room1

admin: Hello from the first tab
admin: Hello from the tab2

Send

Chatroom Page. Image by the author.

In this example, we need to create user groups to locate them in chat rooms. This way, it will be clear which users we will send messages to in a chat room.



Message transmission. Image by the author.

We have the main page for the chat app and a page for each chat room.

```
#urls.py

from django.urls import path
from threechat.views import chat, room

urlpatterns = [
    path('chat/', chat, name='chat_index'),
    path('chat/<str:room_name>/', room, name="chat_room"),
]
```

```
#views.py

from django.shortcuts import render

# Create your views here.
def chat(request):
```

```

        return render(request, './templates/threecat.html', context={})

def room(request, room_name):
    return render(request, './templates/threeroom.html', context={'room_name': room_name})

```

We use *w+* in the *re_path* because it basically matches with anything that comes after *chat/* and it will be recognized and passed to the consumer.

```

#routing.py

from django.urls import re_path
from threecat.consumers import RoomConsumer

ws_urlpatterns = [
    re_path(r'ws/chat/(?P<room_name>\w+)/$', RoomConsumer.as_asgi())
]

```

```

{% include 'base.html' %} {% load static %} {% block content%}
<link rel="stylesheet" href="{% static 'css/chat-style.css' %}?{% now 'U' %}" />
<div>
    <div class="container">
        <div class="row d-flex justify-content-center">
            <div class="col-6">
                <form>
                    <div class="form-group">
                        <label for="textareal" class="h4 pt-5">
                            >Chatroom - {{room_name}}</label>
                        >
                        <textarea class="form-control" id="chat-text" rows="10"></textarea>
                    </div>
                    <div class="form-group">
                        <input class="form-control" id="input" type="text" /><br />
                    </div>
                    <input
                        class="btn btn-success btn-lg btn-block"
                        id="submit"
                        type="button"
                        value="Send"
                    />
                </form>
            </div>
        </div>
    </div>
{% endblock %}

```

```

        </div>
    </div>
</div>
</div>
{{room_name|json_script:"room-name"}}
{{request.user.username|json_script:"username"}}
<script>
    const userName = JSON.parse(document.getElementById("username").textContent);
    const roomName = JSON.parse(document.getElementById("room-name").textContent);
    document.querySelector("#submit").onclick = function (e) {
        const msgInput = document.querySelector("#input");
        const message = msgInput.value;
        chatSocket.send(JSON.stringify({ message: message, username: userName }));
        msgInput.value = "";
    };

    const chatSocket = new WebSocket(
        "ws://" + window.location.host + "/ws/chat/" + roomName + "/"
    );

    chatSocket.onmessage = function (event) {
        const data = JSON.parse(event.data);
        document.querySelector("#chat-text").value +=
            data.username + ": " + data.message + "\n";
    };
</script>
<script
    src="https://code.jquery.com/jquery-3.5.1.slim.min.js"
    integrity="sha384-DfXdz2htPH0lsSSs5nCTpuj/zy4C+OGpamoFVy38MVBnE+IbbVYUew+OrCXaR"
    crossorigin="anonymous"
></script>
<script
    src="https://cdn.jsdelivr.net/npm/popper.js@1.16.1/dist/umd/popper.min.js"
    integrity="sha384-9/reFTGAW83EW2RDu2S0VKAIZap3H66lZ81PoYlFhbGU+6BZp6G7niu735Sk"
    crossorigin="anonymous"
></script>
<script
    src="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.js"
    integrity="sha384-B4gt1jrGC7Jh4AgTPSdUt0Bvf08shuf57BaghqFfPLYxofvL8/KUEfYiJOMMV"
    crossorigin="anonymous"
></script>
{% endblock %}

```

We use this kind of notation (below) to transfer data from the Django side to the JavaScript side. We need *room_name* from the routing (*w+*) and username from the

request.

```
{{room_name|json_script:"room-name"}}  
{{request.user.username|json_script:"username"}}
```

Here we parse the data we get from Django (username) and send it to the WebSocket with the input message that the user enters.

```
document.querySelector("#submit").onclick = function (e) {  
    const msgInput = document.querySelector("#input");  
    const message = msgInput.value;  
    chatSocket.send(JSON.stringify({ message: message, username: userName  
}));msgInput.value = "";  
};
```

We define a WebSocket, similar to what we did in the examples above. When a message is transmitted to it, it parses the data and passes them to the screen.

```

const chatSocket = new WebSocket(
  "ws://" + window.location.host + "/ws/chat/" + roomName + "/"
);

chatSocket.onmessage = function (event) {
  const data = JSON.parse(event.data);
  document.querySelector("#chat-text").value +=
    data.username + ": " + data.message + "\n";
};

```

We define an asynchronous consumer.

```

#consumers.py

import json
from channels.generic.websocket import AsyncWebsocketConsumer

class RoomConsumer(AsyncWebsocketConsumer):
    async def connect(self):
        self.room_name = self.scope['url_route']['kwargs']['room_name']
        self.room_group_name = 'chat_%s' % self.room_name

        await self.channel_layer.group_add(self.room_group_name, self.channel_name)

        await self.accept()

    async def disconnect(self, code):
        await self.channel_layer.group_discard(self.room_group_name, self.channel_name)

    async def receive(self, text_data):
        text_data_json = json.loads(text_data)
        message = text_data_json['message']
        username = text_data_json['username']

        await self.channel_layer.group_send(self.room_group_name, {'type': 'chatroom_message', 'message': message, 'username': username})

    async def chatroom_message(self, event):
        message = event['message']
        username = event['username']

```

```
username = event['username']
await self.send(text_data=json.dumps({'message':message, 'username':username}))
```

Firstly we override the *connect* method.

```
async def connect(self):
    self.room_name = self.scope['url_route']['kwargs']['room_name']
    self.room_group_name = 'chat_%s' % self.room_name

    await self.channel_layer.group_add(self.room_group_name, self.channel_name)

    await self.accept()
```

Let's print *self.scope* to see what has been sent with it:

```
{'type': 'websocket', 'path': '/ws/chat/asd/', 'raw_path': b'/ws/chat/asd/',
'headers': [(b'host', b'127.0.0.1:8000'), (b'pragma', b'no-cache'),
             (b'accept', b'*/*'), (b'sec-websocket-key',
             (b'DtE1JGLUF0e8X2DLld6l6g=='), (b'sec-websocket-version', b'13'),
             (b'accept-language', b'en-US,en;q=0.9'),
             (b'sec-websocket-extensions', b'permessage-deflate'),
             (b'cache-control', b'no-cache'), (b'accept-encoding', b'gzip, deflate'),
             (b'origin', b'http://127.0.0.1:8000'),
             (b'user-agent',
             b'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/605.1.15 (KHTML, like Gecko) Chrome/90.0.4431.59 Safari/604.1'),
             (b'upgrade', b'websocket'), (b'cookie', b'csrftoken=qNTynYDGkiIdYsAZEHLqZwuqv98D3EBMZopHw87e0ENNOavGczQyX280'),
             'query_string': b'',
             'client': ['127.0.0.1', 49417],
             'server': ['127.0.0.1', 8000],
             'subprotocols': [], 'asgi': {'version': '3.0'},
             'cookies': {'csrftoken': 'qNTynYDGkiIdYsAZEHLqZwuqv98D3EBMZopHw87e0ENNOavGczQyX280'},
             'session': <django.utils.functional.LazyObject object at 0x7f9a68617d00>,
             'user': <channels.auth.UserLazyObject object at 0x7f9a483da370>,
```

```
'path_remaining': '', 'url_route': {'args': ()},  
'kwargs': {'room_name': 'asd'}}}]}
```

In the end, we have *kwargs*, we can get the chat room name that the user entered.

We put the *'chat_'* prefix in front of it and create a group and add them to channel layers. Thus, there will be different chat rooms where the same messages are shared. Users will join these different rooms.

Then, the request is accepted and the handshake completes.

We can override the *disconnect* method to discard the chat group from the layers when it is disconnected.

```
async def disconnect(self, code):  
    await self.channel_layer.group_discard(self.room_group_name, self.channel_name)
```

Other than these, we need to override the *receive* method.

```
async def receive(self, text_data):  
    text_data_json = json.loads(text_data)  
    message = text_data_json['message']  
    username = text_data_json['username']  
  
    await self.channel_layer.group_send(self.room_group_name,  
    {'type': 'chatroom_message', 'message': message, 'username': username})
```

It receives a *text_data*:

```
text_data: {"message":"adsasd","username":"admin"}
```

Remember, it's been posted from JavaScript:

```
chatSocket.send(JSON.stringify({ message: message, username: userName }));
```

We parse the data and send it to the chat room. We declare the type as `chatroom_message`, so we create a method with the same name. We dump the message and username.

```
async def chatroom_message(self, event):
    message = event['message']
    username = event['username']
    await self.send(text_data=json.dumps({'message':message, 'username':username}))
```

So, let's try it now. *`python manage.py runserver`*

```
System check identified no issues (0 silenced).
November 29, 2022 - 06:55:23
Django version 4.0, using settings 'examplechannels.settings'
Starting ASGI/Channels version 3.0.4 development server at
http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

`http://127.0.0.1:8000/chat/room1/`

Chatroom - room1

Send

Chatroom page. Image by the author.

First message:

Chatroom - room1

admin: Hello from the first tab

Send

Chatroom page. Image by the author.

Let's open a new tab with the same address and enter a message.

Chatroom - room1

admin: Hello from the tab2

Send

Chatroom page. Image by the author.

If we go back to the first tab, we will see the message from the second tab. The page wasn't refreshed.

Chatroom - room1

admin: Hello from the first tab
admin: Hello from the tab2

Send

Chat room. Image by the author.

Conclusion

Django Channels allow us to develop real-time asynchronous Django applications. It is a leverage that allows us to circumvent Django's synchronous HTTP request restrictions.

I tried to explain the subject with different examples from simple to complex. I hope it was clear. Thank you.

Read More

Discovering Django Forms

Working With Forms in Django

awstip.com