

Django Async

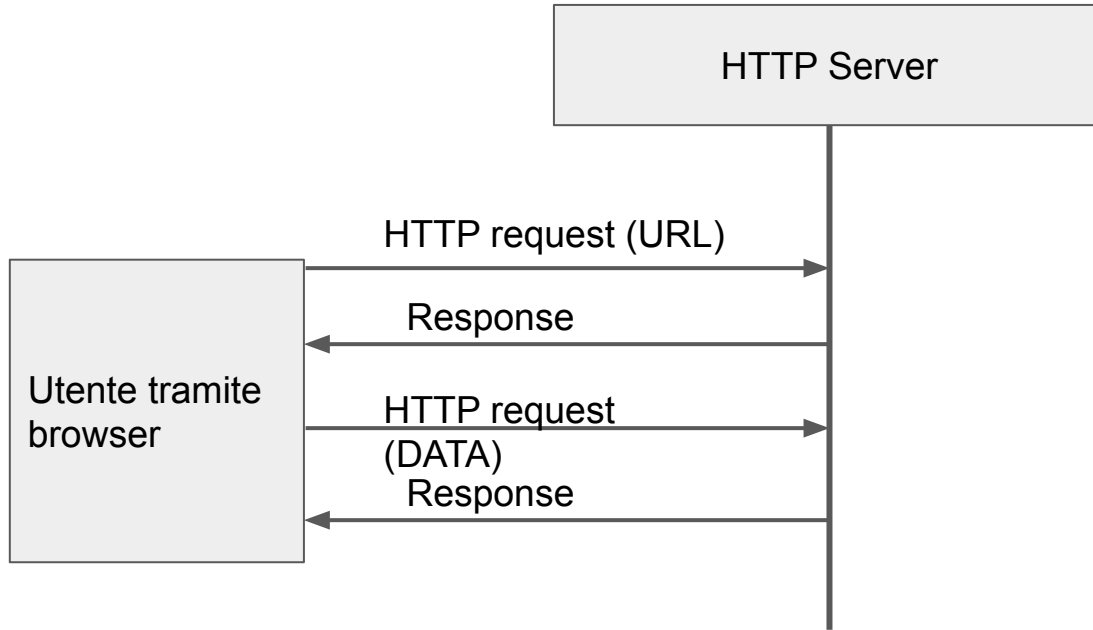
WebSockets & Channels

Come siamo abituati ad usare Django

Protocollo HTTP per richieste

- sincrone
- asincrone tramite JS+AJAX

Schema di funzionamento “classico” HTTP



Problemi?

Ogni coppia request/response si apre e si chiude.

Questo implica:

- Overhead prestazionale nel caso di richieste continue.

Perchè vorremmo poter fare richieste continue?

Le informazioni di nostro interesse sul server potrebbero aggiornarsi continuamente. Dovrei avere un client che periodicamente apre/chiude il ciclo request/response.

Ma abbiamo Ajax...

Mitiga il problema lato presentazione, ma il problema globale lato server rimane...

In altre parole, con Ajax abbiamo visto come possiamo aggiornare il lato presentazione del client tramite richieste asincrone verso il server, senza dover ricaricare la pagina HTML.

Ma è comunque il client a dover chiedere periodicamente al server “hai aggiornamenti?”.

E ad ogni scambio di dati occorre aprire/chiudere una connessione HTTP.

Per certe applicazioni, questo approccio è semplicemente proibitivo.

Si pensi, per esempio ad una chat...

Oltre HTTP...

Abbiamo bisogno di un protocollo **diverso**.

Qualcosa che permetta al client di instaurare una connessione “duratura” ed aperta fino ad una sua **esplicita chiusura**.

Il server è in ascolto ed è conscio delle connessioni aperte, e non si limita a rispondere alle richieste del client, ma può mandare dati legati alla connessione, ma non necessariamente ad una richiesta del client.

Websockets

- Protocollo bidirezionale & Full-Duplex
 - Server e client possono scambiarsi dati in ogni momento
- Supportato da tutti i browser
- In versione sicura (WSS) e non sicura (WS)
- Lato server: pieno supporto alle funzionalità ASGI in aggiunta a WSGI

`ws://`

Websockets: dettagli

Nuovo protocollo basato su TCP

- Opening handshake

- Funzionamento simile all'HTTP (rimane il concetto di request/response, coesiste sulle stesse porte)

- Nuova definizione di “Frame”

- Nuova API in JavaScript per la gestione di messaggi *over ws*

```
var ws = new WebSocket("ws://example.com/foobar");  
ws.onmessage = function(evt) { /* some code */ }  
ws.send("Hello World");  
...
```


Dettagli implementativi lato server

Abbiamo uno o più **consumatori (Consumer)**.

Un *consumer* è un'entità software che è in ascolto di potenziali client.

Un client **apre** una connessione con un consumatore (**open**).

A questo punto si apre un **canale** di comunicazione:

- Il client può mandare messaggi al consumatore (**receive**), e/o
- Il server può mandare messaggi al client (**send**)

Questo accade in ordine arbitrario (definito dalla logica client/server), in maniera iterativa fino alla **chiusura (close)** della connessione

Il server...

Nel nostro caso è un progetto/app scritto con il framework Django.

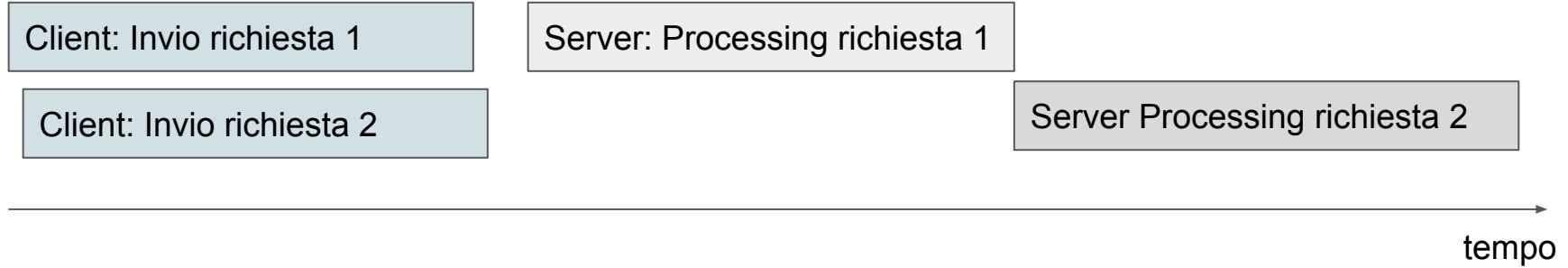
Come già detto nelle lezioni iniziali, django tradizionalmente supportava solo il protocollo WSGI, ma nelle versioni più moderne supporta anche ASGI.

Abbiamo già parlato di ASGI vs. WSGI quando abbiamo discusso dei costrutti paralleli/concorrenti di python...

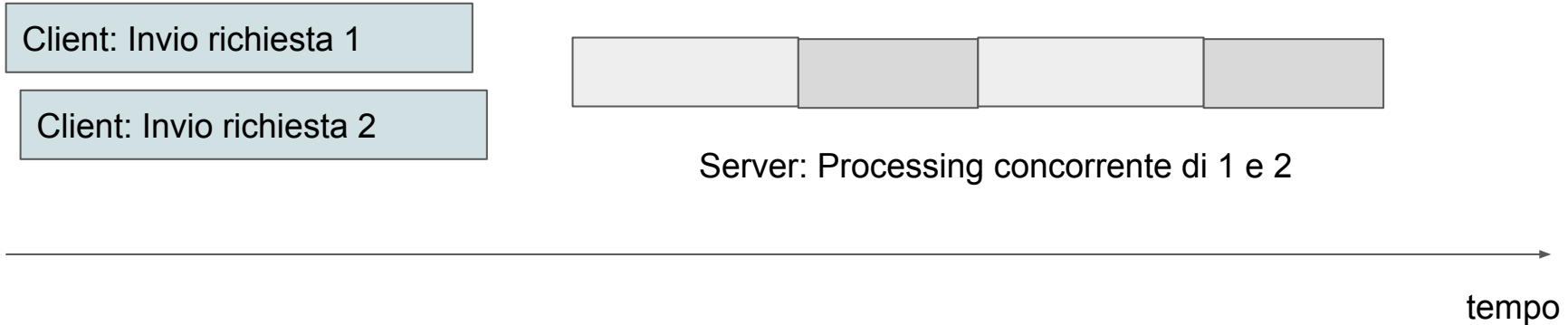
Questo ci consente una buona dose di flessibilità quando si tratta di dover gestire tante richieste da diversi client che a loro volta instaurano connessioni su websocket potenzialmente persistenti.

ASGI vs WSGI

GESTIONE TRAMITE WSGI



GESTIONE TRAMITE ASGI



WSGI == sempre “Processing Serializzato”?

Non necessariamente, possiamo comunque servire le richieste in thread paralleli.

Per questo, esistono WSGI servers production-ready da poter facilmente “affiancare” a django.

<https://docs.djangoproject.com/en/4.0/howto/deployment/wsgi/>

Il caso di studio: una chat

Si crei un progetto django, con tanto di app.

Tale progetto deve essere in grado di servire le richieste HTTP come siamo abituati a fare. Quindi restituendo in maniera dinamica pagine HTML.

In una di queste pagine HTML, possiamo inserire codice javascript per gestire una semplice chat tramite i websocket.

Con semplice s'intende niente DB/modelli/tabelle e nessuna gestione di permessi ed autenticazione. Solo un botta e risposta tra diversi client "anonimi".

Il progetto di esempio

si trova nel git del sito del corso in

.../django/django_chchat/...

progetto “chat”

L'app si chiama “chat_app”

django-channels

```
pipenv install channels
```

Channels include i seguenti packages:

- **Channels**, i canali integrati con Django
- **Daphne**, un server HTTP e per Websocket
- **asgiref**, una libreria ASGI
- **channels_redis**, un gestore di canali production-ready, che noi non useremo.

django-channels: il minimo per usare i websocket

- **Channels**, i canali integrati con Django
- ~~**Daphne**, un server HTTP e per WebSocket~~
- **asgiref**, una libreria ASGI
- ~~channels_redis, un gestore di canali production-ready, che noi non useremo.~~

Completamento dell'installazione

In settings.py:

```
INSTALLED_APPS = [  
    'django.contrib.admin', 'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions', 'django.contrib.messages',  
    'django.contrib.staticfiles',  
    <"nome applicazione chat">,  
    'channels'  
]  
  
WSGI_APPLICATION = 'chat.wsgi.application'  
  
ASGI_APPLICATION = 'chat.asgi.application'
```

Migrate e runserver

System check identified no issues (0 silenced).

May 10, 2022 - 09:41:58

Django version 4.0.4, using settings 'chat.settings'

**Starting ASGI/Channels version 3.0.4 development server at
http://127.0.0.1:8000/**

Quit the server with CTRL-BREAK.

Negli altri progetti...

System check identified no issues (0 silenced).

May 10, 2022 - 09:45:51

Django version 4.0.4, using settings 'biblio3.settings'

Starting development server at http://127.0.0.1:8000/

Quit the server with CTRL-BREAK.

E' cambiato il development server!

Quello attivato con il comando runserver attiva un **development** server in grado di gestire le richieste secondo l'approccio ASGI.

Rimane un development server, quindi **non da usare in produzione**, ma il pacchetto channels ci permette di usare anche **daphne**.

<https://channels.readthedocs.io/en/stable/deploying.html>

Torniamo alla nostra chatapp

Cosa occorre fare:

- 1) Stabilire le regole di routing.
- 2) Implementare il Consumer
- 3) ...FBV & CBV a piacere...
 - a) Lato HTML, inseriremo nel template uno script in js usando l'API per i websocket

Routing

Regole di “instradamento”.

Il nostro client adesso deve poter:

- Mandare una richiesta HTTP al server, il quale risponde con una FBV/CBV
 - Questo lo sappiamo già fare
- Mandare una richiesta su protocollo ws al server, il quale risponde con la logica arbitraria costruita a livello di **Consumer**.

in chat/asgi.py

```
import os

from django.core.asgi import get_asgi_application

from channels.routing import ProtocolTypeRouter, URLRouter

from channels.auth import AuthMiddlewareStack

from .routing import ws_urlpatterns

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

#application = get_asgi_application() #questo va commentato...

application = ProtocolTypeRouter(

    {

        "http" : get_asgi_application(),

        "websocket" : AuthMiddlewareStack(URLRouter(ws_urlpatterns))

    }

)
```

ProtocolTypeRouter

Ci permette di definire regole di routing a partire dal root project.

In particolare, ci permette di stabilire come protocolli diversi debbano essere serviti.

Per esempio: ad una richiesta HTTP corrisponde un normale web app, tipo quelle che conosciamo. I diversi endpoint seguono le regole degli url patterns.

Se invece la richiesta arriva su protocollo ws, allora dovremo aggiungere un livello di routing, in cui specificheremo i diversi endpoint che punteranno a diversi consumers.

AuthMiddlewareStack

Ci permette di interfacciarsi con il sistema di autenticazione di Django tramite i consumers attivati su websockets.

In altre parole. Noi sappiamo come funziona Auth in Django, e per esempio sappiamo come ottenere informazioni riguardanti sessioni/users etc... da un FBV/CBV...

Dato che un consumer non è esattamente una view, ci occorre un middleware che ci permetta di accedere a sessions/cookies/users in maniera simile a come lo faremmo in una view. In particolare, l'AuthMiddlewareStack fornisce una variabile “scope”, in sola lettura che noi possiamo usare a questo scopo.

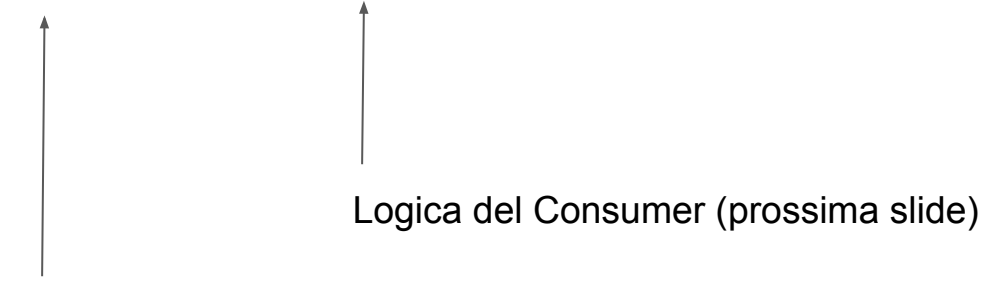
URLRouter

“Traduce” gli url per gli endpoint dei nostri websocket consumers in regole di routing.

Infatti, la variabile `ws_urlpatterns` la faremo noi, in un file chiamato `routing.py` e somiglia a quello che abbiamo fatto fino ad ora con gli `urlpatterns`.

in routing.py (file da creare)

```
ws_urlpatterns = [  
    path("ws/chatws/", WSConsumerChat.as_asgi()),  
]
```



endpoint/punto di accesso al servizio di chat
tramite WS

Logica del Consumer (prossima slide)

Consumer

Ci sono diversi tipi. Noi creeremo un **AsyncWebsocketConsumer** importato da `channels.generic.websocket`.

Il consumatore dovrà accettare le connessioni dai client.

Mettersi in ascolto sui loro messaggi, spedire i messaggi ai client.

Implementazione semplificata: il consumer accetterà sempre le connessioni in ingresso. Il consumers salva tutti i messaggi dei client in una struttura dati condivisa (no DB per comodità) e, sempre su richiesta, spedirà tale struttura dati ai client che periodicamente ne faranno richiesta...

La struttura dati condivisa

E' una lista di messaggi.

Un messaggio è un dizionario-json-like in cui esiste la chiave “user” e la chiave “msg”.

Sempre per comodità, vediamo questa lista come un buffer da gestire.

Raggiunti i 20 elementi, essa viene “ripulita”.

in consumers.py (da creare)

```
import json
```

```
messages_list = []
```

```
class WSConsumerChat(AsyncWebsocketConsumer):
```

```
    async def connect(self):
```

```
        await self.accept()
```

```
        await self.send("SERVER: Eccoti connesso!")
```

```
    async def receive(self, text_data=None, bytes_data=None):
```

```
        if text_data == "UPDATE":
```

```
            stot = ""
```

```
            for m in messages_list:
```

```
                stot += m["user"] + ": " + m["msg"] + "\n"
```

```
            await self.send(stot)
```

```
        else:
```

```
            if text_data != None:
```

```
                messages_list.append(json.loads(text_data))
```

```
    if len(messages_list) > 20:
```

```
        messages_list.clear()
```

Da notare

Tutti i metodi sono ereditati e riscritti dal padre.

Tali metodi devono essere definiti per essere operazioni async.

Accettare una connessione, spedire un messaggio etc... sono tutte operazioni bloccanti che possono eseguire concorrentemente tra i diversi clients. Quindi siamo costretti ad usare **await**.

Esempio: mentre accetti una connessione di un client, puoi mandare un messaggio ad un altro...

Lato Client

Abbiamo comunque bisogno di una pagina HTML “normale”.

Alla quale poi inseriremo logica arbitraria tramite JavaScript.

Tale HTML sarà sottoforma di DTL, caricato dinamicamente in seguito al raggiungimento di una FBV...

Quindi in `templates/chatapp/chatpage.html`

L'HTML/DTL

```
{% extends "base.html" %}

{% block title %} Chat Page {% endblock %}

{% block content %}

<h1>    {{msg}}    </h1>


    <label for="uname">Username:</label>
    <input type="text" name="uname" id="username"> <br> <br>
    <label for="msg">Messaggio:</label>
    <input type="text" name="msg" id="msg"> <br> <br>

    <p> Chatlog: </p>

    <textarea id="chatlog" rows="20" cols="50">
    </textarea>

<br>|
    <button onclick="btnClick()">Chatta</button>
```

ChatPage Room!

Username:

Messaggio:

Chatlog:

SERVER: Eccoti connesso!

Chatta

Lo script

```
var socket = new WebSocket('ws://127.0.0.1:8000/ws/chatws/');

socket.onmessage = function(event){

    var data = event.data;
    var d = document.querySelector('#chatlog');
    d.value = data + '\r\n';

}

function btnClick() {

    var obj = new Object();
    obj.user = document.querySelector('#username').value;
    obj.msg = document.querySelector('#msg').value;
    var string = JSON.stringify(obj);
    socket.send(string);
    document.querySelector('#username').disabled = true
    document.querySelector('#msg').value = "";

}

var intervalId = setInterval(function() {
    socket.send("UPDATE")
}, 1);
```

stabilito in routing.py

callback di ricezione (riceve una stringa)

Invio del messaggio tramite JSON

Messaggio di richiesta di aggiornamenti periodici...

Raggiungibilità e logica della view:

```
#in urls.py
```

```
urlpatterns = [  
    path('admin/', admin.site.urls),  
    path('chat/', include("chatapp.urls"))  
]
```

```
#in chatapp/urls.py
```

```
urlpatterns = [  
    path("", chatpage, name="chatpage"),  
]
```

```
#in chatapp/views.py
```

```
def chatpage(request):  
    return render(request, "chatapp/chatpage.html", context={"msg": "ChatPage  
Room!"})
```

ChatPage Room!

Username:

Messaggio:

Chatlog:

Utente1: Ciao

Utente2: Ciao anche a te.

Utente1: Come va?

Utente2: Mah, le solite cose...

ChatPage Room!

Username:

Messaggio:

Chatlog:

Utente1: Ciao

Utente2: Ciao anche a te.

Utente1: Come va?

Utente2: Mah, le solite cose...

Questa chat...

Funziona.

Ma ha diversi potenziali problemi...

Il primo, la gestione dei dati “condivisi”. Estremamente semplicistica, manca un lock alla risorsa `messages_list`...

Il secondo, per fare questa cosa **i websocket sono strettamente necessari?**

Il modo in cui client e server interagiscono ricorda molto l'esempio di autocompletamento visto con JS+AJAX, **ma uno dei vantaggi di WS su AJAX è quello di avere un server in grado di mandare messaggi al client senza dover necessariamente ricevere una richiesta da un utente connesso...**

Chat V2: chat done right!

Facciamo una versione migliorata della chat.

Questa volta, delegheremo a django/channels la gestione dei messaggi dei client.

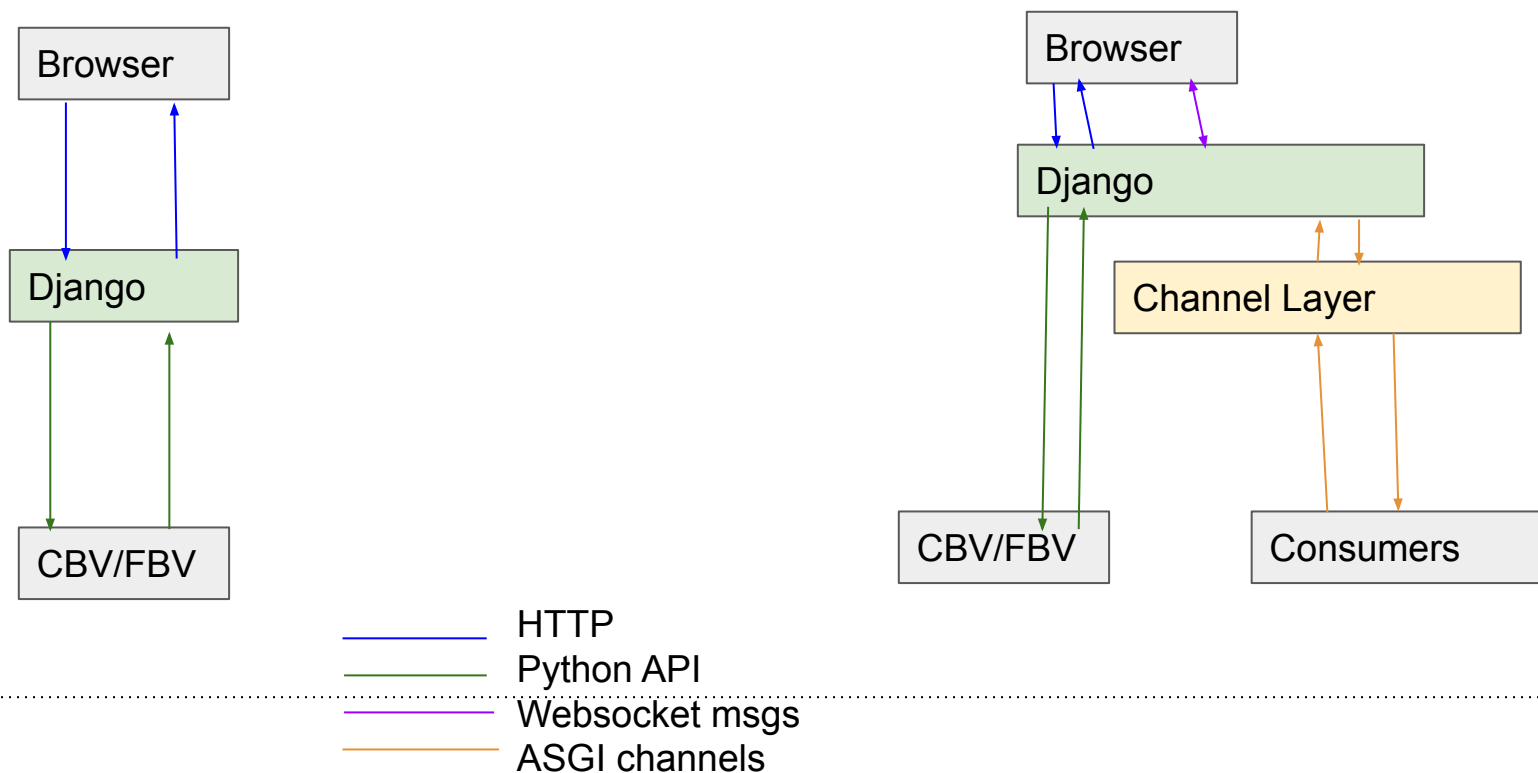
Non avremo una pagina di chat, ma diverse “stanze”.

Ciascuna stanza è rappresentata da una mailbox.

Il server, tramite il consumer gestirà le mailbox in maniera asincrona: quando l'utente A manderà un messaggio, tale messaggio verrà mandato in broadcast a tutti gli utenti presenti nella stanza, senza che essi debbano richiedere aggiornamenti periodici.

Channels & Channel layer

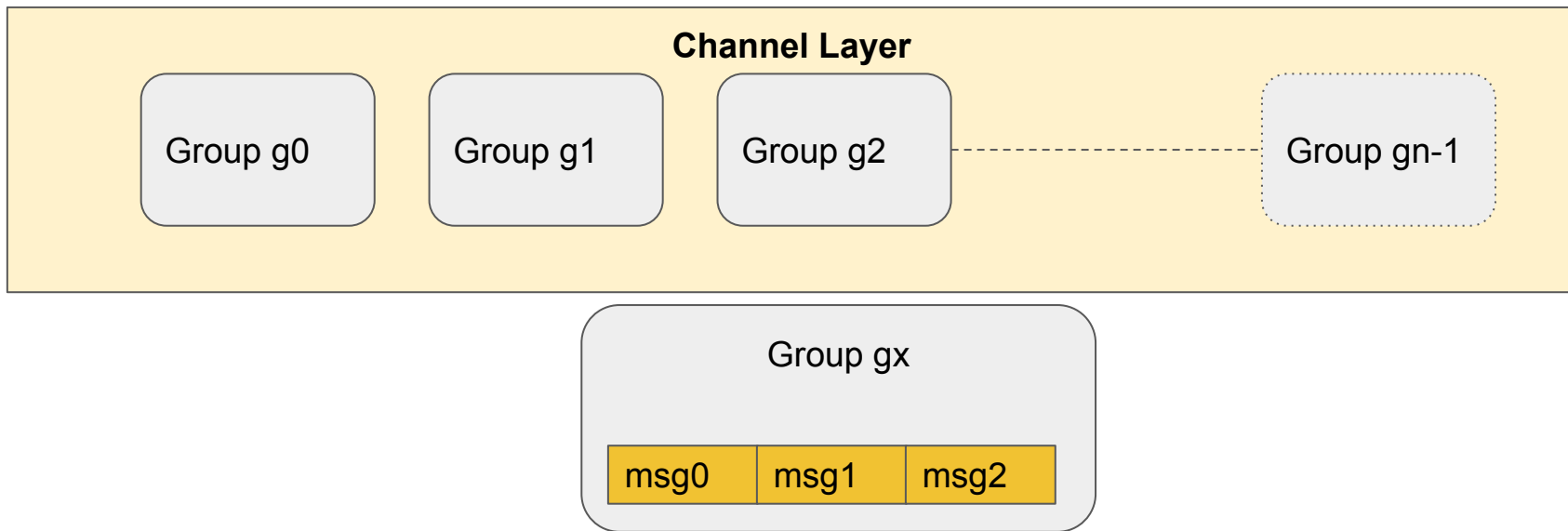
I canali rappresentano le nostre mailbox. Ironicamente, la versione 1 della chat non li usava, sebbene i Consumers “ereditassero” dal package “channels”.



Channel layer

Racchiude le strutture dati di comunicazione e le divide in gruppi.

Ciascun gruppo è una coda di messaggi. Un consumer può mandare un messaggio ad un channel group. Tale messaggio è accodato. I messaggi accodati vengono poi spediti a tutti i consumers “iscritti” a quel gruppo.



Channels

Un canale è quindi il flusso di comunicazione tra i consumer ed il channel layer.

Si crea indicando a quale gruppo si vuole iscrivere il consumer, dato che ciascun gruppo è indicato con un nome.

Se il gruppo indicato dal nome non esiste, esso viene creato alla prima connessione del consumer tramite richiesta ws lato client.

Alla disconnessione del client, il **canale** associato consumer-gruppo si cancella, ma il **gruppo** rimane.

Questi gruppi non hanno nulla a che fare con i Django Groups visti in Auth.

Dettagli implementativi dei channel layers

L'implementazione dei channel layer è trasparente dal punto di vista del programmatore. Quindi potremmo non sapere che tipo di strutture dati sono utilizzati per salvare i messaggi, o eventuali limiti di occupazione in memoria etc...

Il package channels di django ci mette a disposizione due implementazioni di channel layers:

- in memory channels
 - Debug/development only
- redis-channels
 - Production ready

https://channels.readthedocs.io/en/stable/topics/channel_layers.html

Setup

Dobbiamo dire a Django quale channel backend (implementazione) utilizzare:

In settings.py:

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

Altre modifiche: accessibilità

routing.py

```
ws_urlpatterns = [  
    path("ws/chatws/", WSConsumerChat.as_asgi()),  
    path("ws/chatws/<str:room>/", WSConsumerChatChannels.as_asgi())  
]
```

chatapp/urls.py

```
urlpatterns = [ path("", chatpage, name="chatpage"),  
    path("<str:room>/", chatroom, name="chatroom")  
]
```

chatapp/views.py

```
def chatroom(request, room):  
    return render(request, "chatapp/chatpage2.html", context={"msg": room})
```

Il consumer (WSConsumerChatChannels in consumers.py)

Connessione/Disconnessione/gestione canali e gruppi:

```
class WSConsumerChatChannels(AsyncWebsocketConsumer):  
  
    async def connect(self):  
        self.room_name = self.scope['url_route']['kwargs']['room']  
        self.room_group_name = 'chat_' + self.room_name  
  
        await self.channel_layer.group_add(  
            self.room_group_name,  
            self.channel_name  
        )  
  
        await self.accept()  
  
    async def disconnect(self, close_code):  
        await self.channel_layer.group_discard(  
            self.room_group_name,  
            self.channel_name  
        )
```

Il consumer (WSConsumerChatChannels in consumers.py)

Gestione messaggi

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

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

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

    await self.send(text_data=json.dumps({
        'msg': message,
        'user': username,
    })))
```

Il consumer (WSConsumerChatChannels in consumers.py)

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

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

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

    await self.send(text_data=json.dumps({
        'msg': message,
        'user': username,
    })))
```

parsing del messaggio in ingresso in formato json

Messaggio girato alla mailbox del gruppo precedentemente creato in funzione del canale

Definizione di callback e dati in ingresso da scatenarsi all'accodamento del messaggio.

Il client

Da un punto di vista logico e funzionale, non cambia quasi nulla.

Di certo non abbiamo bisogno di un timed event in cui mandare una richiesta di update.

la callback “onmessage” dell’oggetto di tipo WebSocket è dove andremo ad aggiornare il log della chat.

Abbiamo quindi un file chatapp/templates/chatapp/chatpage2.html che è il nostro template. Lo abbiamo “abbellito” un po usando bootstrap...

In particolare, ad ogni messaggio ricevuto **creiamo** un elemento HTML di tipo “media object”, definito tra le classi di Bootstrap v4.

Lato DTL\HTML

```
<p> Chatlog: </p>

<div class="container mt-3" id="chatlog">

  <!--
  | https://www.w3schools.com/bootstrap4/bootstrap\_media\_objects.asp
  |-->

</div>

<br><br>
<label for="uname">Username:</label>
<input type="text" name="uname" id="username"> <br> <br>
<label for="msg">Messaggio:</label>
<input type="text" name="msg" id="msg"> <br> <br>


<button onclick="btnClick()">Chatta</button>
```

Lato JS, andrò ad aggiungere “figli” al componente con id=”chatlog”, usando la DOM manipulation


localhost:8000/chat/TecnologieWeb/

TecnologieWeb


Chatlog:



utente1 13:50:45
Ciao! Tu cosa porti come progettone di tecn. web?



utente2 13:51:3
Pensavo di metterci una chat real-time!



utente1 13:51:11
...e se poi te ne penti?


Username:

Messaggio:


localhost:8000/chat/TecnologieWeb/

TecnologieWeb


Chatlog:



utente1 13:50:45
Ciao! Tu cosa porti come progettone di tecn. web?



utente2 13:51:3
Pensavo di metterci una chat real-time!



utente1 13:51:11
...e se poi te ne penti?

Username:

Messaggio:

TecnologieWeb

Chatlog:



utente3 13:53:1

Di cosa state parlando?

Username:

Messaggio:

Chatta

Chatlog:



utente1 13:50:45

Ciao! Tu cosa porti come progettone di tecn. web?



utente2 13:51:3

Pensavo di metterci una chat real-time!



utente1 13:51:11

...e se poi te ne penti?



utente3 13:53:1

Di cosa state parlando?

Username:

Messaggio:

Chatta

Chatlog:



utente1 13:50:45

Ciao! Tu cosa porti come progettone di tecn. web?



utente2 13:51:3

Pensavo di metterci una chat real-time!



utente1 13:51:11

...e se poi te ne penti?



utente3 13:53:1

Di cosa state parlando?

Username:

Messaggio:

Chatta

```

var socket = new WebSocket('ws://127.0.0.1:8000/ws/chatws/' + document.querySelector("#header").innerHTML + '/');

var roomUsersColors = {};

socket.onmessage = function(event){

    var container = document.querySelector("#chatlog");
    var data = JSON.parse(event.data);

    if(roomUsersColors[data["user"]]==undefined)
        roomUsersColors[data["user"]] = Math.random() * 360;

    var chatMsg = document.createElement("div");
    chatMsg.className = "media border p-3";
    var img = document.createElement("img");
    img.src = "http://127.0.0.1:8000/static/imgs/img_avatar3.png";
    img.alt = data["user"];
    img.className = "mr-3 mt-3 rounded-circle";
    img.style = "width:60px;filter:hue-rotate(" + roomUsersColors[data["user"]] + "deg);";
    var innerDiv = document.createElement("div");
    innerDiv.className = "media-body";

    var ts = new Date();
    innerDiv.innerHTML = "
    <h4>
    " + data["user"] + "
    <small><i>
    " + ts.getHours() + ":" + ts.getMinutes() + ":" + ts.getSeconds() + "
    </i></small></h4>
    " +
    "<p>
    " + data["msg"] + "
    </p>";

    chatMsg.appendChild(img);
    chatMsg.appendChild(innerDiv);
    container.appendChild(chatMsg);

}

```

Non ci sono
variazioni nella
funzione di invio
messaggio tramite
ws rispetto alla
versione1 della chat!

```
var img = document.createElement("img");  
img.src = "http://127.0.0.1:8000/static/imgs/img_avatar3.png";  
img.alt = data["user"];  
img.className = "mr-3 mt-3 rounded-circle"  
img.style = "width:60px;filter:hue-rotate("+ roomUsersColors[data["user"]] +"deg);";
```

```
var innerDiv = document.createElement("div");  
innerDiv.className = "media-body";  
  
var ts = new Date();  
innerDiv.innerHTML = "<h4>" + data["user"] + " <small><i>" +  
                      ts.getHours()+":"+ts.getMinutes()+":"+ts.getSeconds() + " </i></small></h4>" +  
                      "<p>" + data["msg"] + "</p>";
```



utente1 14:11:36

Corpo del messaggio



utente2 14:11:48

Altro messaggio...