What will we build ?

This book will focus on how to create a real world mobile chat application with **Ionic 5 / Angular 12, Capacitor 3 and Django 3**. The **Chat** application will manage private chat between two persons and is not a group chat application.

For this application, we can list these main functionalities :

- Login / Register / Forget password screens
- List of chats screen with search user features
- Chat screen : real time message with WebSocket, loading previous messages, sending attachments
- Receiving push notifications

So this is what this book is about: We will learn each steps required to create a real world chat mobile application with it's associated backend in a day to day process.

On backend, we will use Django and Django channels to have a socket based chat and we will also learn how to save and retrieve chat history.

On frontend, we will develop a Ionic application with angular framework, which will communicate with our backend to manage the chats. we will deploy the application on mobile devices using Capacitor.

Ready to start ?

Day 1 : setup and design Backend

First we will focus on the backend development before diving into the Ionic part.

You can find the source code for this day one journey in GitHub

Setup : Software required

Before diving into the subject of this book, we need to install software and set up our environment for the backend part. If you are familiar with Django development you can skip this section.

Python 3

All the development will be done with **python 3** so i will assume that python 3 is already install on your computer. If not, you will find tutorials on the internet on how to install it.

Pip

PIP is the Python Package Installer and is really helpful to install Django and all the libraries required to develop any **Django** project.

If you are on a Mac just open a terminal and enter

```
sudo easy_install pip
```

Virtualenv

Virtualenv is a tool to create isolated python environment. It is a recommended and standard way to develop Django projetcs. So let's begin. First go in a directory where you would like to create the backend.

```
cd Programmation
mkdir ChatTuto
mkdir ChatTuto/Backend
mkdir ChatTuto/Frontend
```

I create a **ChatTuto** directory and inside this directory, i create the **Backend** in which we will develop our backend and a **Frontend** directory in which we will develop the Ionic application.

To install virtualenv

```
sudo pip install virtualenv
```

And now we create the virtual environment into a directory named **venv**

```
virtualenv -p python3 venv
```

Once installed, we need to activate the environment using the command:

```
source venv/bin/activate
```

Django

We can install Django 3

```
pip install Django==3
```

And now we can create our django backend which will be called **chattuto**

```
django-admin startproject chattuto
cd chattuto
```

Ok now we can begin to implement our models that will be used to deal with our **Chat application**.

Conception

For each mobile app that you need to develop, the process is always the same. Starting from the storyboard of the application, we can deduce which entities will be needed. From these entities, we can conceptualize the models, the back-office (web administration) required to manage data and finally as last step, design the API which will be consumed by the frontend application.

> *Usually, i don't recommend to implement the API without having seen application storyboards. Because it is better to implement API endpoints that will fulfill each screen requirements and then improve drastically the performance. If you don't proceed that way, to provide data to a screen, then may be you will need to make multiple http requests whereas only one request could have done the job.*

For our **Chat** application, we need to display a list of chats and messages inside a choosen chat. And of course a chat is between two users, so we can easily deduce the following models:

- User
- Chat
- Message

This is a good start to design our Django models.

Designing models

We need to create our first app in Django to declare our models and then we will add a backoffice which will be used by the administrator of the mobile application. To create this **chat** application in Django:

```
python manage.py startapp chat
```

Inside the newly created **chat** directory, Django will initialize some standard files for us. At this stage, we are interested with the **models.py** file.

> *I didn't mentioned earlier but i use PyCharm to develop all my Django projects. You will find a community edition (FREE to use) or a PRO edition. But you can also use Eclipse with PyDev plugin, or VSCode. It's really up to you.*

User model

Most of mobile application have users who can subscribe or login, so this will be the first entity we need. By default, Django includes a **User** model entity and all required methods to deals with authentification.

But by default in a Django project, the field used to manage authentication is a **username** whereas for a mobile application it makes more sense to use an **email** for authentication process.

However it is possible to easily modify the Django default mechanism to use an email. To do so we have to extend the default **User** entity.

So let's do it:

```
class UserManager(BaseUserManager):
    use_in_migrations = True

    def _create_user(self, email, password, **extra_fields):
        """
        Creates and saves a User with the given email and password.
        """
        if not email:
            raise ValueError('The given email must be set')
        email = self.normalize_email(email)
        user = self.model(email=email, **extra_fields)
        user.set_password(password)
        user.save(using=self._db)
        return user

    def create_user(self, email, password=None, **extra_fields):
        extra_fields.setdefault('is_superuser', False)
        return self._create_user(email, password, **extra_fields)

    def create_superuser(self, email, password, **extra_fields):
        extra_fields.setdefault('is_superuser', True)

        if extra_fields.get('is_superuser') is not True:
            raise ValueError('Superuser must have is_superuser=True.')

        return self._create_user(email, password, **extra_fields)

class User(AbstractBaseUser, PermissionsMixin):
    id = models.UUIDField(primary_key=True, default=uuid.uuid4, editable=False)
    email = models.EmailField(_('email address'), unique=True)
    first_name = models.CharField(_('first name'), max_length=30, blank=True)
    last_name = models.CharField(_('last name'), max_length=30, blank=True)
    date_joined = models.DateTimeField(_('date joined'), auto_now_add=True)
    is_active = models.BooleanField(_('active'), default=True)
    is_staff = models.BooleanField(_('active'), default=True)
    avatar = models.ImageField(upload_to='avatars/', null=True, blank=True)
    lastConnexionDate = models.DateTimeField(null=True, blank=True)
    valid = models.BooleanField(default=True)

    objects = UserManager()

    USERNAME_FIELD = 'email'
    REQUIRED_FIELDS = []

    class Meta:
        verbose_name = _('user')
        verbose_name_plural = _('users')
    createdAt = models.DateTimeField(auto_now_add=True)
    updatedAt = models.DateTimeField(auto_now=True)

    @property
    def last_login(self):
        return self.lastConnexionDate

    def __str__(self):
        return u'%s' % (self.email)
```

Now **User** can be authenticated with email because we set

```
USERNAME_FIELD = 'email'
```

> *The **lastConnexionDate** can be used to track last connexion of the user into the mobile application.*

Chat models

A chat is a dialog conversation between two users. The dialog is composed of a list of text messages but it also can be something else than text such as photos, videos, ... We can deduce that we will need two more models to store Chats and Messages.

```
class Chat(models.Model):
    id = models.UUIDField(primary_key=True, default=uuid.uuid4, editable=False)
    fromUser = models.ForeignKey(User, db_index=True,on_delete=models.SET_NULL,
null=True,related_name="fromuser")
    toUser = models.ForeignKey(User, db_index=True,on_delete=models.SET_NULL,
null=True,related_name="toUser")
    createdAt = models.DateTimeField(auto_now_add=True)
    updatedAt = models.DateTimeField(auto_now=True)
    unique_together = (("fromUser", "toUser"),)

    def __str__(self):
        return u'%s - %s' % (self.fromUser,self.toUser)

class Message(models.Model):
    id = models.UUIDField(primary_key=True, default=uuid.uuid4, editable=False)
    refChat = models.ForeignKey(Chat, db_index=True,on_delete=models.CASCADE)
    message = models.TextField()
    msg_type = (
        (0, "TEXT"),
        (1, "GEOLOC"),
        (2, "PHOTO"),
    )
    type = models.IntegerField(choices=msg_type, default=0)
    extraData = models.CharField(default='', null=True, blank=True, max_length=255)
    author = models.ForeignKey(User,
db_index=True,related_name='author',on_delete=models.SET_NULL,null=True)
    isRead = models.BooleanField(default=False)
    createdAt = models.DateTimeField(auto_now_add=True)
    updatedAt = models.DateTimeField(auto_now=True)

    def __str__(self):
        return u'%s - %d' % (self.refChat,self.type)
```

> *A chat must be unique between two users so we set a **unique_together** key based on users primary keys.*

> *A message refers to a chat, can be a text, a geolocation position (using latitude/longitude) or a photo. We could extend this message type based on the application requirements like videos, sound... And of course a message has an author (the **User** who wrotes the message)*

Setup database

To create our models, we need to install a database. I will use **PostgreSQL** with **postgis** extension.

On Mac, you will need **Postres** software and you can use **Postico** on your local machine while developping. It's quite easy to use and install.

On Windows or Linux, you will have to setup your database which can be quite painful. To connect to your database, you can use the free **DBeaver** software.

I will assume that this step is done and will move forward on to next section.

> *I strongly recommend to use **Clever cloud** services, a french provider that i will use for these tutorials. With **Clever cloud** services, you can focus on your development features and not architecture. They offer a FREE database hosting for development. It's like Heroku but cheaper and more performant.*

Configure Django project

We can configure our Django project to use our database. First we will need the **psycopg2** package which is the PostgreSQL python driver. Let's add it to our **requirements.txt** file

```
Django==3.1.7
pillow==8.0.1
psycopg2==2.8.5 --no-binary psycopg2
```

I'm using the non binary version to be sure that the driver will be accurate with the server system libraries (it was a recommendation from the Clever cloud team).

> *As you could have notice, i'm always specifiying the version of the library that i want to use. If you don't do that while deploying your project on a server, the latest version of libraries will be installed and could lead to a bug (if something changed in a library). Specifying the version will avoid that.*

Don't forget to install the libraries

```
pip install -r requirements.txt
```

Now, we can configure **Django** to use our database. We edit the **settings.py** file inside the **chattuto** directory and will replace the **DATABASES** section with:

```

AUTH_USER_MODEL = "chat.User"

# Database
# https://docs.djangoproject.com/en/3.0/ref/settings/#databases
POSTGRESQL_ADDON_URI = os.getenv("POSTGRESQL_ADDON_URI")
POSTGRESQL_ADDON_PORT = os.getenv("POSTGRESQL_ADDON_PORT")
POSTGRESQL_ADDON_HOST = os.getenv("POSTGRESQL_ADDON_HOST")
POSTGRESQL_ADDON_DB = os.getenv("POSTGRESQL_ADDON_DB")
POSTGRESQL_ADDON_PASSWORD = os.getenv("POSTGRESQL_ADDON_PASSWORD")
POSTGRESQL_ADDON_USER = os.getenv("POSTGRESQL_ADDON_USER")
REDIS_URL= os.getenv("REDIS_URL")
DATABASES = {
    'default': {
        'ENGINE': 'django.contrib.gis.db.backends.postgis',
        'NAME': POSTGRESQL_ADDON_DB,
        'USER': POSTGRESQL_ADDON_USER,
        'PASSWORD': POSTGRESQL_ADDON_PASSWORD,
        'HOST': POSTGRESQL_ADDON_HOST,
        'PORT': POSTGRESQL_ADDON_PORT,
        'CONN_MAX_AGE': 1200,
    }
}
```

I'm getting my database connexion parameters from environment variables and then set the databases dictionary keys with these values.

> *Using environment variables it's easy to switch from a development database to a production database. Values are not hardcoded.*

> *Please notice the line **AUTH_USER_MODEL = "chat.User"**. Since we override the default Django User entity, this line is very important and tells Django that it needs to use our custom model.*

With **PyCharm** you can declare your environment variables in the settings of your project, or you can use a **environment.env** file and use this plugin to declare them, which is more easy.

Within my **environment.env** file, i declare my database connection parameters like this:

```
POSTGRESQL_ADDON_URI=postgresql://u6rogoiqgpupp3mvhqi9:F1Kc6J1pKEt5X28iX4Fn@bx8gb6agpf58jh3rtupa-
postgresql.services.clever-cloud.com:5432/bx8gb6agpf58jh3rtupa
POSTGRESQL_ADDON_PORT=5432
POSTGRESQL_ADDON_HOST=bx8gb6agpf58jh3rtupa-postgresql.services.clever-cloud.com
POSTGRESQL_ADDON_DB=b9pgqzgccizfpklha4gv
POSTGRESQL_ADDON_USER=u6rogoiqgpupp3mvhqi9
POSTGRESQL_ADDON_PASSWORD=<YOURPASSWORD>
DEBUG=True
MEDIA_URL_PREFIX=/media
STATIC_URL_PREFIX=/static
```

> *Don't forget to replace these values with your own database connexion values*

We could also do the same for the **DEBUG** variable

```
DEBUG = os.getenv("DEBUG")
```

and add the DEBUG value in our **environment.env** file:

```
DEBUG=True
```

We can try to run our **Django** project with PyCharm and if database connexion is OK, we should see the following:

```
Performing system checks...

System check identified no issues (0 silenced).

You have 17 unapplied migration(s). Your project may not work properly until you apply the
migrations for app(s): admin, auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
August 07, 2020 - 11:24:32
Django version 3.0, using settings 'chattuto.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

To launch **Django** with command line, you will need to export your environment variables and then run in a terminal:

```
python manage.py runserver
```

> *To export my environment variables, i usually use a text file and then source it (on Mac or Linux).*

Create database tables

At this point, **Django** tells us that we need to run migrations. You can learn more about migrations here but basically it means that we have modifications in our **models.py** file that are not reflected on the database which is true since we haven't created it yet.

First we need to edit again the **settings.py** file to add our application **chat** to the **INSTALLED_APPS** dictionnary:

```
# Application definition
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'chat'
]
```

Now because we have overrided the default **Django User**, we first need to make a migrations with our **chat** application (which will tell Django where to find the **User** model) and then we can migrate to create all other tables required by Django. So let's do it by typing in a terminal:

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

Implement Chat server with Websockets and django channels

To implement our Chat backend we will use websockets which are more performant than HTTP requests and are asynchronous. To implement websockets within Django, **Channels** has been introduce recently. You can read more about the definition on the channel website documentation.

Let's add channel and daphne to our **requirements.txt** file

```
Django==3.1.7
pillow==8.0.1
psycopg2==2.8.5 --no-binary psycopg2
daphne==3.0.1
channels
channels_redis
```

and install it

```
pip install -r requirements.txt
```

From now and to finish this day one tutorial, we will just follow the Channel Tutorials part 1 to 2. We will stop at tutorial two, because we don't want our chat server to be asynchronous.

Before diving into that, we will create a superuser for our Django backend:

```
python manage.py createsuperuser
```

and we will implement our **admin.py** file to be able to manage Chat or Message models thru the admin:

```
from django.contrib import admin

from chat.models import *


class ChatAdmin(admin.ModelAdmin):
    fieldsets = [
        (None, {'fields': ['fromUser','toUser']}),
    ]
    list_display = ('fromUser','toUser','createdAt','updatedAt',)
    search_fields =
('fromUser__email','toUser__email','fromUser__last_name','toUser__last_name',)
    raw_id_fields = ('fromUser','toUser',)
    list_select_related = ('fromUser','toUser',)


class MessageAdmin(admin.ModelAdmin):
    fieldsets = [
        (None, {'fields': ['refChat','message','author','isRead','type','extraData']}),
    ]
    list_display = ('refChat','message','author','isRead','createdAt',)
    ordering = ('-createdAt',)
    search_fields = ('message','author__email',)
    raw_id_fields = ('author',)
    list_select_related = ('refChat','author',)

# Register your models here.
admin.site.register(Chat,ChatAdmin)
admin.site.register(Message,MessageAdmin)
```

> *NOTE FOR THE TCE, we can refer to the course i have written about the Django admin*

Ok so now as i said we will just follow the **Channel tutorials** to learn basics on Django channels.

Adding an index view

Inside our **chat** directory, let's create a template directory containing a **chat** directory

```
mkdir templates
mkdir templates/chat
```

Then we create an **index.html** file with the code

```html
<!-- chat/templates/chat/index.html -->
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8"/>
    <title>Chat Rooms</title>
</head>
<body>
    What chat room would you like to enter?<br>
    <input id="room-name-input" type="text" size="100"><br>
    <input id="room-name-submit" type="button" value="Enter">

    <script>
        document.querySelector('#room-name-input').focus();
        document.querySelector('#room-name-input').onkeyup = function(e) {
            if (e.keyCode === 13) {  // enter, return
                document.querySelector('#room-name-submit').click();
            }
        };

        document.querySelector('#room-name-submit').onclick = function(e) {
            var roomName = document.querySelector('#room-name-input').value;
            window.location.pathname = '/chat/' + roomName + '/';
        };
    </script>
</body>
</html>
```

This page will let us specify wich **chat** room we would like to join. Let's write the Django view which will render this template. Edit the **views.py** file inside the **chat** directory (not the one from the template directory but from the Django chat application)

```python
# chat/views.py
from django.shortcuts import render

def index(request):
    return render(request, 'chat/index.html')
```

Then we need to create an url for this view. Edit or create an **urls.py** file if it doesn't exists, and write

```python
# chat/urls.py
from django.urls import path

from . import views

urlpatterns = [
    path('', views.index, name='index'),
]
```

The next step is to point the root URLconf at the chat.urls module. In **chattuto/urls.py**, add an import for django.conf.urls.include and insert an include() in the urlpatterns list, so you have:

```
from django.conf.urls import include
from django.urls import path
from django.contrib import admin

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

Now we can launch the server

```
python manage runserver
```

And go to the following url : http://127.0.0.1:8000/chat/

You should see the text “What chat room would you like to enter?” along with a text input to provide a room name.

Adding the channel library

Now it's time to integrate Channels.

Let's start by creating a root routing configuration for Channels. A Channels routing configuration is an ASGI application that is similar to a Django URLconf, in that it tells Channels what code to run when an HTTP request is received by the Channels server.

Start by adjusting the **chattuto/asgi.py** file to include the following code:

```
# chattuto/asgi.py
import os

from channels.auth import AuthMiddlewareStack
from channels.routing import ProtocolTypeRouter, URLRouter
from django.core.asgi import get_asgi_application
import chat.routing

os.environ.setdefault("DJANGO_SETTINGS_MODULE", "chattuto.settings")

application = ProtocolTypeRouter({
  "http": get_asgi_application(),
  "websocket": AuthMiddlewareStack(
        URLRouter(
            chat.routing.websocket_urlpatterns
        )
    ),
})
```

Now add the Channels library to the list of installed apps. Edit the **chattuto/settings.py** file and add 'channels' to the INSTALLED_APPS **settings.py** file

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'channels',
    'chat'
]
```

We also need to point Channels at the root routing configuration. Edit the **chattuto/settings.py** file again and add the following line:

```
# Channels
ASGI_APPLICATION = 'chattuto.asgi.application'
```

With Channels now in the installed apps, it will take control of the runserver command, replacing the standard Django development server with the Channels development server.

```
python manage runserver
```

And we will see in the console logs :

```
Django version 3.1.7, using settings 'chattuto.settings'
Starting ASGI/Channels version 3.0.4 development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

The important line is **Starting ASGI/Channels**

Adding the room view

Create a new file chat/templates/chat/room.html

```
<!-- chat/templates/chat/room.html -->
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8"/>
    <title>Chat Room</title>
</head>
<body>
    <textarea id="chat-log" cols="100" rows="20"></textarea><br>
    <input id="chat-message-input" type="text" size="100"><br>
    <input id="chat-message-submit" type="button" value="Send">
    {{ room_name|json_script:"room-name" }}
    <script>
        const roomName = JSON.parse(document.getElementById('room-name').textContent);

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

        chatSocket.onmessage = function(e) {
            const data = JSON.parse(e.data);
            document.querySelector('#chat-log').value += (data.message + '\n');
        };

        chatSocket.onclose = function(e) {
            console.error('Chat socket closed unexpectedly');
        };

        document.querySelector('#chat-message-input').focus();
        document.querySelector('#chat-message-input').onkeyup = function(e) {
            if (e.keyCode === 13) {  // enter, return
                document.querySelector('#chat-message-submit').click();
            }
        };

        document.querySelector('#chat-message-submit').onclick = function(e) {
            const messageInputDom = document.querySelector('#chat-message-input');
            const message = messageInputDom.value;
            chatSocket.send(JSON.stringify({
                'message': message
            }));
            messageInputDom.value = '';
        };
    </script>
</body>
</html>
```

Then we can add the view function for the room view in **chat/views.py**

```
# chat/views.py
from django.shortcuts import render

def index(request):
    return render(request, 'chat/index.html')

def room(request, room_name):
    return render(request, 'chat/room.html', {
        'room_name': room_name
    })
```

And create the route for the room view in **chat/urls.py**:

```
# chat/urls.py
from django.urls import path

from . import views

urlpatterns = [
    path('', views.index, name='index'),
    path('<str:room_name>/', views.room, name='room'),
]
```

We can start the server again :

```
python manage runserver
```

Go to http://127.0.0.1:8000/chat/ in your browser and to see the index page.

Type in “lobby” as the room name and press enter. You should be redirected to the room page at http://127.0.0.1:8000/chat/lobby/ which now displays an empty chat log.

Type the message “hello” and press enter. Nothing happens. In particular the message does not appear in the chat log. Why?

The room view is trying to open a WebSocket to the URL ws://127.0.0.1:8000/ws/chat/lobby/ but we haven’t created a consumer that accepts WebSocket connections yet. If you open your browser’s JavaScript console, you should see an error that looks like:

```
WebSocket connection to 'ws://127.0.0.1:8000/ws/chat/lobby/' failed: Unexpected response code:
500
```

Creating a consumer

When Django accepts an HTTP request, it consults the root URLconf to lookup a view function, and then calls the view function to handle the request. Similarly, when Channels accepts a WebSocket connection, it consults the root routing configuration to lookup a consumer, and then calls various functions on the consumer to handle events from the connection.

We will write a basic consumer that accepts WebSocket connections on the path /ws/chat/ROOM_NAME/ that takes any message it receives on the WebSocket and echos it back to the same WebSocket.

Create a new file **chat/consumers.py**

```
# chat/consumers.py
import json
from channels.generic.websocket import WebsocketConsumer

class ChatConsumer(WebsocketConsumer):
    def connect(self):
        self.accept()

    def disconnect(self, close_code):
        pass

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

        self.send(text_data=json.dumps({
            'message': message
        }))
```

We need to create a routing configuration for the chat app that has a route to the consumer. Create a new file **chat/routing.py**:

```
from django.urls import re_path

from . import consumers

websocket_urlpatterns = [
    re_path(r'ws/chat/(?P<room_name>\w+)/$', consumers.ChatConsumer.as_asgi()),
]
```

Let’s verify that the consumer for the **/ws/chat/ROOM_NAME/** path works. Run migrations to apply database changes (Django’s session framework needs the database) and then start the Channels development server:

```
python manage.py migrate
python manage.py runserver
```

Go to the room page at http://127.0.0.1:8000/chat/lobby/ which now displays an empty chat log.

Type the message “hello” and press enter. You should now see “hello” echoed in the chat log.

However if you open a second browser tab to the same room page at http://127.0.0.1:8000/chat/lobby/ and type in a message, the message will not appear in the first tab. For that to work, we need to have multiple instances of the same ChatConsumer be able to talk to each other. Channels provides a channel layer abstraction that enables this kind of communication between consumers.

A channel layer is a kind of communication system. It allows multiple consumer instances to talk with each other, and with other parts of Django.

We will use **REDIS** has a channel layer. Once again i will use **Clever cloud** to setup a Redis server really easily.

Then i just need to update the **channel layer** config in my **chattutu/settings.py** file:

```
REDIS_URL= os.getenv("REDIS_URL")
# Channels
ASGI_APPLICATION = 'chattuto.asgi.application'
CHANNEL_LAYERS = {
    'default': {
        'BACKEND': 'channels_redis.core.RedisChannelLayer',
        'CONFIG': {
            "hosts": [REDIS_URL],
        },
    },
}
```

and declare a new environment variable based on the value provided by the **Clever cloud** add-on:

```
export REDIS_URL='redis://:<YourPASSWORD>@btka0rdhzdplteztka9i-redis.services.clever-
cloud.com:3067'
```

Now we can edit the **consumers.py* file again and replace the existing code with:

```
# chat/consumers.py
import json
from asgiref.sync import async_to_sync
from channels.generic.websocket import WebsocketConsumer

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

        # Join room group
        async_to_sync(self.channel_layer.group_add)(
            self.room_group_name,
            self.channel_name
        )

        self.accept()

    def disconnect(self, close_code):
        # Leave room group
        async_to_sync(self.channel_layer.group_discard)(
            self.room_group_name,
            self.channel_name
        )

    # Receive message from WebSocket
    def receive(self, text_data):
        text_data_json = json.loads(text_data)
        message = text_data_json['message']

        # Send message to room group
        async_to_sync(self.channel_layer.group_send)(
            self.room_group_name,
            {
                'type': 'chat_message',
                'message': message
            }
        )

    # Receive message from room group
    def chat_message(self, event):
        message = event['message']

        # Send message to WebSocket
        self.send(text_data=json.dumps({
            'message': message
        }))
```

When a user posts a message, a JavaScript function will transmit the message over WebSocket to a ChatConsumer. The ChatConsumer will receive that message and forward it to the group corresponding to the room name. Every ChatConsumer in the same group (and thus in the same room) will then receive the message from the group and forward it over WebSocket back to JavaScript, where it will be appended to the chat log.

And voila.

> *Remember we use the the **Channel tutorials** to learn basics on Django channels so please refer to it for more information or clarifications.*

You can find the source code for this day one journey in GitHub

In next tutorial **Day two** we will learn how to modify our **Channel** to use our models **Chat** and **Message** and to implement persistent messages. We will also learn how to secure our **Chat server** with authentication and we will setup an API with **Django Rest Framework**.

Questions / Answers

1. What are Django models ?

> *Models are classed which will represent your entities and will be used to create the database tables. 2. By default, which field is using Django for authenticating users : Username or email ? Username 3. What is the library name to create web sockets with Django ? Django Channels 4. What do we have to setup to enable multiple asynchronous connection with Django Channels ? A channel layer*