Tutorial Part 2: Implement a Chat Server

This tutorial begins where Tutorial 1 left off. We'll get the room page working so that you can chat with yourself and others in the same room.

Add the room view

We will now create the second view, a room view that lets you see messages posted in a particular chat room.

Create a new file `chat/templates/chat/room.html`. Your app directory should now look like:

```
chat/
    __init__.py
    templates/
        chat/
            index.html
            room.html
    urls.py
    views.py
```

Create the view template for the room view in `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>
```

Create 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})
```

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"),
]
```

Start the Channels development server:

```
$ python3 manage.py 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
```

Write your first 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.

> **Note**
>
> It is good practice to use a common path prefix like `/ws/` to distinguish WebSocket connections from ordinary HTTP connections because it will make deploying Channels to a production environment in certain configurations easier.
>
> In particular for large sites it will be possible to configure a production-grade HTTP server like nginx to route requests based on path to either (1) a production-grade WSGI server like Gunicorn+Django for ordinary HTTP requests or (2) a production-grade ASGI server like Daphne+Channels for WebSocket requests.
>
> Note that for smaller sites you can use a simpler deployment strategy where Daphne serves all requests - HTTP and WebSocket - rather than having a separate WSGI server. In this deployment configuration no common path prefix like `/ws/` is necessary.

Create a new file `chat/consumers.py`. Your app directory should now look like:

```
chat/
    __init__.py
    consumers.py
    templates/
        chat/
            index.html
            room.html
    urls.py
    views.py
```

Put the following code in `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}))
```

This is a synchronous WebSocket consumer that accepts all connections, receives messages from its client, and echos those messages back to the same client. For now it does not broadcast messages to other clients in the same room.

> **Note**
>
> Channels also supports writing *asynchronous* consumers for greater performance. However any asynchronous consumer must be careful to avoid directly performing blocking operations, such as accessing a Django model. See the Consumers reference for more information about writing asynchronous consumers.

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`. Your app directory should now look like:

```
chat/
    __init__.py
    consumers.py
    routing.py
    templates/
        chat/
            index.html
            room.html
    urls.py
    views.py
```

Put the following code in `chat/routing.py`:

```
# 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()),
]
```

We call the `as_asgi()` classmethod in order to get an ASGI application that will instantiate an instance of our consumer for each user-connection. This is similar to Django's `as_view()`, which plays the same role for per-request Django view instances.

(Note we use `re_path()` due to limitations in URLRouter.)

The next step is to point the main ASGI configuration at the **chat.routing** module. In `mysite/asgi.py`, import `AuthMiddlewareStack`, `URLRouter`, and `chat.routing`; and insert a `'websocket'` key in the `ProtocolTypeRouter` list in the following format:

```
# mysite/asgi.py
import os

from channels.auth import AuthMiddlewareStack
from channels.routing import ProtocolTypeRouter, URLRouter
from channels.security.websocket import AllowedHostsOriginValidator
from django.core.asgi import get_asgi_application

os.environ.setdefault("DJANGO_SETTINGS_MODULE", "mysite.settings")
# Initialize Django ASGI application early to ensure the AppRegistry
# is populated before importing code that may import ORM models.
django_asgi_app = get_asgi_application()

import chat.routing

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

This root routing configuration specifies that when a connection is made to the Channels development server, the `ProtocolTypeRouter` will first inspect the type of connection. If it is a WebSocket connection (**ws://** or **wss://**), the connection will be given to the `AuthMiddlewareStack`.

The `AuthMiddlewareStack` will populate the connection's **scope** with a reference to the currently authenticated user, similar to how Django's `AuthenticationMiddleware` populates the **request** object of a view function with the currently authenticated user. (Scopes will be discussed later in this tutorial.) Then the connection will be given to the `URLRouter`.

The `URLRouter` will examine the HTTP path of the connection to route it to a particular consumer, based on the provided `url` patterns.

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
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying auth.0010_alter_group_name_max_length... OK
  Applying auth.0011_update_proxy_permissions... OK
  Applying auth.0012_alter_user_first_name_max_length... OK
  Applying sessions.0001_initial... OK
$ python3 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.

Go to the terminal where you ran the `runserver` command and press Control-C to stop the server.

Enable a channel layer

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.

A channel layer provides the following abstractions:

- A **channel** is a mailbox where messages can be sent to. Each channel has a name. Anyone who has the name of a channel can send a message to the channel.
- A **group** is a group of related channels. A group has a name. Anyone who has the name of a group can add/remove a channel to the group by name and send a message to all channels in the group. It is not possible to enumerate what channels are in a particular group.

Every consumer instance has an automatically generated unique channel name, and so can be communicated with via a channel layer.

In our chat application we want to have multiple instances of `ChatConsumer` in the same room communicate with each other. To do that we will have each ChatConsumer add its channel to a group whose name is based on the room name. That will allow ChatConsumers to transmit messages to all other ChatConsumers in the same room.

We will use a channel layer that uses Redis as its backing store. To start a Redis server on port 6379, run the following command:

```
$ docker run -p 6379:6379 -d redis:5
```

We need to install channels_redis so that Channels knows how to interface with Redis. Run the following command:

```
$ python3 -m pip install channels_redis
```

Before we can use a channel layer, we must configure it. Edit the `mysite/settings.py` file and add a `CHANNEL_LAYERS` setting to the bottom. It should look like:

```
# mysite/settings.py
# Channels
ASGI_APPLICATION = "mysite.asgi.application"
CHANNEL_LAYERS = {
    "default": {
        "BACKEND": "channels_redis.core.RedisChannelLayer",
        "CONFIG": {
            "hosts": [("127.0.0.1", 6379)],
        },
    },
}
```

Note

It is possible to have multiple channel layers configured. However most projects will just use a single `'default'` channel layer.

Let's make sure that the channel layer can communicate with Redis. Open a Django shell and run the following commands:

```
$ python3 manage.py shell
>>> import channels.layers
>>> channel_layer = channels.layers.get_channel_layer()
>>> from asgiref.sync import async_to_sync
>>> async_to_sync(channel_layer.send)('test_channel', {'type': 'hello'})
>>> async_to_sync(channel_layer.receive)('test_channel')
{'type': 'hello'}
```

Type Control-D to exit the Django shell.

Now that we have a channel layer, let's use it in `ChatConsumer`. Put the following code in `chat/consumers.py`, replacing the old code:

```
# 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.

Several parts of the new `ChatConsumer` code deserve further explanation:

- `self.scope["url_route"]["kwargs"]["room_name"]`

    - Obtains the `'room_name'` parameter from the URL route in `chat/routing.py` that opened the WebSocket connection to the consumer.
    - Every consumer has a scope that contains information about its connection, including in particular any positional or keyword arguments from the URL route and the currently authenticated user if any.
- `self.room_group_name = "chat_%s" % self.room_name`
    - Constructs a Channels group name directly from the user-specified room name, without any quoting or escaping.
    - Group names may only contain alphanumerics, hyphens, underscores, or periods. Therefore this example code will fail on room names that have other characters.
- `async_to_sync(self.channel_layer.group_add)(...)`
    - Joins a group.
    - The `async_to_sync(...)` wrapper is required because ChatConsumer is a synchronous WebsocketConsumer but it is calling an asynchronous channel layer method. (All channel layer methods are asynchronous.)
    - Group names are restricted to ASCII alphanumerics, hyphens, and periods only and are limited to a maximum length of 100 in the default backend. Since this code constructs a group name directly from the room name, it will fail if the room name contains any characters that aren't valid in a group name or exceeds the length limit.
- `self.accept()`
    - Accepts the WebSocket connection.
    - If you do not call `accept()` within the `connect()` method then the connection will be rejected and closed. You might want to reject a connection for example because the requesting user is not authorized to perform the requested action.
    - It is recommended that `accept()` be called as the *last* action in `connect()` if you choose to accept the connection.
- `async_to_sync(self.channel_layer.group_discard)(...)`
    - Leaves a group.
- `async_to_sync(self.channel_layer.group_send)`
    - Sends an event to a group.
    - An event has a special `'type'` key corresponding to the name of the method that should be invoked on consumers that receive the event.

Let's verify that the new consumer for the `/ws/chat/ROOM_NAME/` path works. To start the Channels development server, run the following command:

```
$ python3 manage.py runserver
```

Open a browser tab to the room page at http://127.0.0.1:8000/chat/lobby/. Open a second browser tab to the same room page.

In the second browser tab, type the message “hello” and press enter. You should now see “hello” echoed in the chat log in both the second browser tab and in the first browser tab.

You now have a basic fully-functional chat server!

This tutorial continues in Tutorial 3.