Tutorial Part 3: Rewrite Chat Server as Asynchronous

This tutorial begins where Tutorial 2 left off. We'll rewrite the consumer code to be asynchronous rather than synchronous to improve its performance.

Rewrite the consumer to be asynchronous

The `ChatConsumer` that we have written is currently synchronous. Synchronous consumers are convenient because they can call regular synchronous I/O functions such as those that access Django models without writing special code. However asynchronous consumers can provide a higher level of performance since they don't need to create additional threads when handling requests.

`ChatConsumer` only uses async-native libraries (Channels and the channel layer) and in particular it does not access synchronous Django models. Therefore it can be rewritten to be asynchronous without complications.

Note

Even if `ChatConsumer` *did* access Django models or other synchronous code it would still be possible to rewrite it as asynchronous. Utilities like asgiref.sync.sync_to_async and channels.db.database_sync_to_async can be used to call synchronous code from an asynchronous consumer. The performance gains however would be less than if it only used async-native libraries.

Let's rewrite `ChatConsumer` to be asynchronous. Put the following code in `chat/consumers.py`:

```python
# chat/consumers.py
import json

from channels.generic.websocket import AsyncWebsocketConsumer


class ChatConsumer(AsyncWebsocketConsumer):
    async 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
        await self.channel_layer.group_add(self.room_group_name, self.channel_name)

        await self.accept()

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

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

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

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

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

This new code is for ChatConsumer is very similar to the original code, with the following differences:

- `ChatConsumer` now inherits from `AsyncWebsocketConsumer` rather than `WebsocketConsumer`.
- All methods are `async def` rather than just `def`.
- `await` is used to call asynchronous functions that perform I/O.
- `async_to_sync` is no longer needed when calling methods on the channel layer.

Let's verify that the consumer for the `/ws/chat/ROOM_NAME/` path still 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.

Now your chat server is fully asynchronous!

This tutorial continues in Tutorial 4.