Clases y Métodos Relacionados con WebSockets y Conexiones Asíncronas

En el contexto del proyecto, se utilizan clases y métodos relacionados con WebSockets y conexiones asíncronas, principalmente a través de Django Channels. A continuación, se explica en detalle la funcionalidad de cada método relevante, cómo se relacionan con las clases heredadas y cómo interactúan con tecnologías como Redis y Docker.

1 Clase WebSocketConsumer

La clase WebSocketConsumer es proporcionada por Django Channels y permite manejar conexiones WebSocket. Esta clase incluye métodos básicos que se pueden sobrescribir para personalizar el comportamiento.

- `connect(self)`:
  - Se ejecuta cuando el cliente intenta establecer una conexión WebSocket.
  - Aquí se puede autenticar al usuario, asignarlo a un grupo (sala de chat) y aceptar la conexión.
  - Ejemplo en `[ChatConsumer] [C:\Users\Alesa\Desktop\App Chat\mywebsite\chat\cons`

```
def connect(self):
    self.id =
       self.scope['url_route']['kwargs']['room_id']
    self.room_group_name = f'sala_chat_{self.id}'
    async_to_sync(self.channel_layer.group_add)(
        self.room_group_name,
        self.channel_name
    )
    self.accept()
```

    - `self.scope`: Contiene información del contexto de la conexión (como el usuario autenticado o la URL).
    - `self.accept()`: Acepta la conexión WebSocket.
    - `self.channel_layer.group_add`: Añade el canal del cliente a un grupo (sala de chat).

- `disconnect(self, close_code)`:
  - Se ejecuta cuando el cliente cierra la conexión o el servidor la finaliza.
  - Util para limpiar recursos, eliminar al usuario del grupo y notificar a otros usuarios.
  - Ejemplo en `[ChatConsumer] [C:\Users\Alesa\Desktop\App Chat\mywebsite\chat\cons`

```
def disconnect(self, close_code):
    async_to_sync(self.channel_layer.group_discard)(
        self.room_group_name,
        self.channel_name
    )
```

- `receive(self, text_data)`:
  - Maneja los mensajes enviados por el cliente a través del WebSocket.
  - Se utiliza para procesar datos, validar información y enviar respuestas.
  - Ejemplo en [ChatConsumer] [C:\Users\Alesa\Desktop\App Chat\mywebsite\chat\cons

```python
def receive(self, text_data):
    text_data_json = json.loads(text_data)
    message = text_data_json['message']
    async_to_sync(self.channel_layer.group_send)(
        self.room_group_name,
        {
            'type': 'chat_message',
            'message': message,
            'username': self.user.username,
        }
    )
```

- `chat_message(self, event)`:
  - Método personalizado que se ejecuta cuando se recibe un mensaje del grupo.
  - Envia el mensaje a todos los clientes conectados excepto al remitente.
  - Ejemplo en [ChatConsumer] [C:\Users\Alesa\Desktop\App Chat\mywebsite\chat\cons

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

2 Clase AsyncWebsocketConsumer

Aunque no se utiliza directamente en este proyecto, es una alternativa asíncrona a WebSocketConsumer. Permite manejar conexiones WebSocket de forma completamente asíncrona sin necesidad de usar async_to_sync.

- Dado que WebSocketConsumer es síncrono, pero las operaciones con el channel_layer son asíncronas, se utiliza async_to_sync para convertir funciones asíncronas en funciones síncronas.
- `async_to_sync(self.channel_layer.group_add)`: Convierte la operación asíncrona de agregar un canal a un grupo en una operación síncrona.
- `async_to_sync(self.channel_layer.group_send)`: Convierte la operación asíncrona de enviar un mensaje a un grupo en una operación síncrona.

3 Uso de Redis como Channel Layer

Redis se utiliza como backend para el Channel Layer, que es la capa de comunicación entre los consumidores y los grupos. Permite que los mensajes se envíen entre diferentes instancias del servidor.

- **Configuración**: En `[settings.py]` `[C:\Users\Alesa\Desktop\App Chat\mywebsite\mywebs` se configura Redis como el backend del Channel Layer:

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

- **Docker para Redis**: Redis puede ejecutarse en un contenedor Docker para facilitar su configuración y despliegue:

```
docker run -d -p 6379:6379 --name redis redis
```

4 Integración con ASGI

Django Channels reemplaza el servidor WSGI por ASGI para manejar conexiones asíncronas. Esto se configura en `[asgi.py]` [C:\Users\Alesa\Desktop\App Chat\mywebsite\mywebsite\asg

- **Middleware de Autenticación**:

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

  - `ProtocolTypeRouter`: Ruta diferentes tipos de conexiones (HTTP, WebSocket, etc.) al middleware correspondiente.

5 Comunicación Cliente-Servidor

En el cliente, se utiliza JavaScript para interactuar con el WebSocket:

- **Conexión al WebSocket**:

```
var chatSocket = new WebSocket('ws://' +
   window.location.host + '/ws/room/' + room_id + '/');
```

- **Enviar Mensajes**:

```
chatSocket.send(JSON.stringify({
    'type': 'chat_message',
    'message': message_value.trim(),
}));
```

- **Recibir Mensajes**:

```
chatSocket.onmessage = function(e) {
    const data = JSON.parse(e.data);
    if (data.type === 'chat_message') {
        console.log(data.message);
    }
};
```

6 Resumen

- WebSocketConsumer maneja conexiones WebSocket.
- `connect`, `disconnect` y `receive` son métodos personalizados como `chat_message`.
- Backend para el Channel Layer, utilizando para comunicación entre consumidores y grupos.
- Docker: Facilita la ejecución de Redis.
- ASGI: Reemplaza WSGI para manejar conexiones asíncronas.

Este diseño permite manejar chats en tiempo real de manera eficiente y escalable.