Channels学习

519021910025 钟睿哲

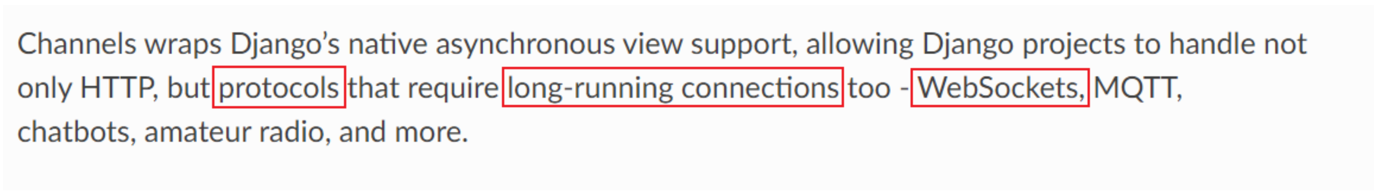
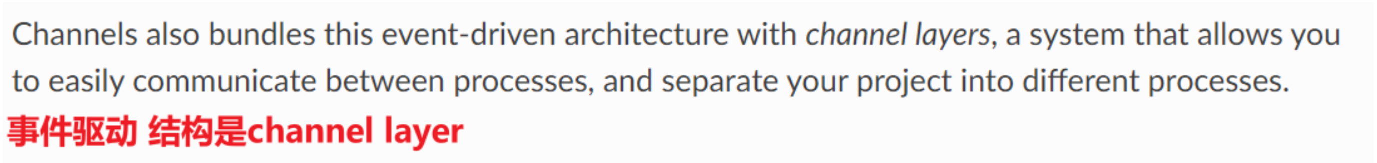
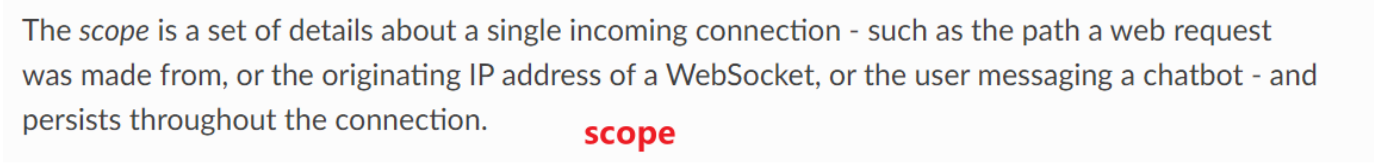
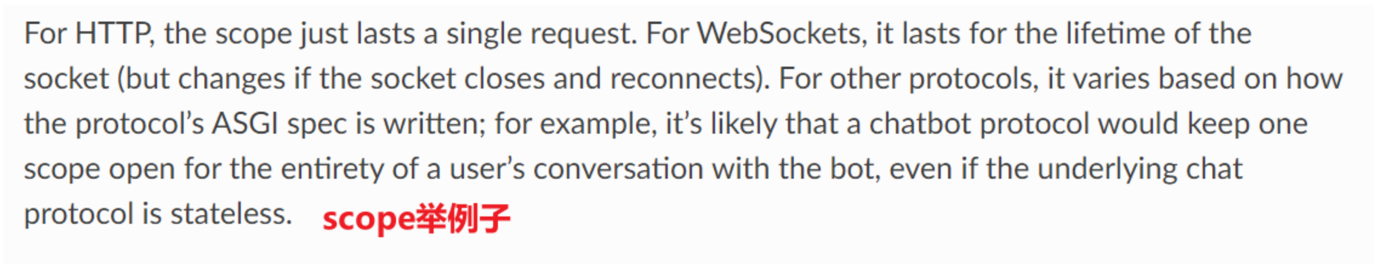
简介

- Channels用作实时通信，可以用于创建一个聊天室

官方教程

- Channels
- Docker
- 80端口被占用

笔记截图

- 
  Channels wraps Django's native asynchronous view support, allowing Django projects to handle not only HTTP, but protocols that require long-running connections too - WebSockets, MQTT, chatbots, amateur radio, and more.
- 允许Django处理HTTP之外的协议，比如WebSockets等等，可以粗浅地理解为异步通信的协议
- 
  Channels also bundles this event-driven architecture with *channel layers*, a system that allows you to easily communicate between processes, and separate your project into different processes.
  事件驱动 结构是channel layer
- 
  The *scope* is a set of details about a single incoming connection - such as the path a web request was made from, or the originating IP address of a WebSocket, or the user messaging a chatbot - and persists throughout the connection.
  scope
- 由scope和event构成
- scope包含了connection的基本信息
- 
  For HTTP, the scope just lasts a single request. For WebSockets, it lasts for the lifetime of the socket (but changes if the socket closes and reconnects). For other protocols, it varies based on how the protocol's ASGI spec is written; for example, it's likely that a chatbot protocol would keep one scope open for the entirety of a user's conversation with the bot, even if the underlying chat protocol is stateless.
  scope举例子
-

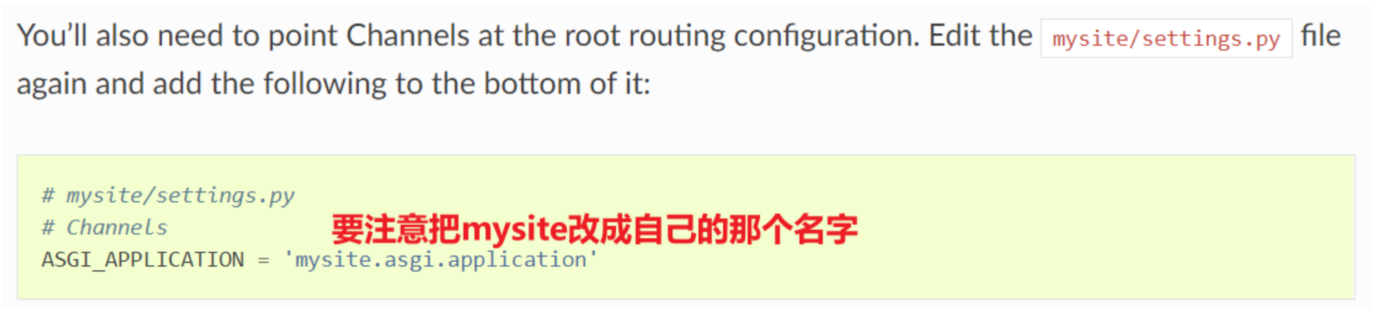
During the lifetime of this *scope*, a series of *events* occur. These represent user interactions - making a HTTP request, for example, or sending a WebSocket frame. Your Channels or ASGI applications will be **instantiated once per scope**, and then be fed the stream of *events* happening within that scope to decide what to do with. **events和scope的关系**

- An example with HTTP:

  - The user makes an HTTP request.
  - We open up a new `http` type scope with details of the request's path, method, headers, etc.
  - We send a `http.request` event with the HTTP body content
  - The Channels or ASGI application processes this and generates a `http.response` event to send back to the browser and close the connection.
  - The HTTP request/response is completed and the scope is destroyed.
- Use HTTP as an example
- An example with a chatbot:

  - The user sends a first message to the chatbot.
  - This opens a scope containing the user's username, chosen name, and user ID.
  - The application is given a `chat.received_message` event with the event text. It does not have to respond, but could send one, two or more other chat messages back as `chat.send_message` events if it wanted to.
  - The user sends more messages to the chatbot and more `chat.received_message` events are generated.
  - After a timeout or when the application process is restarted the scope is closed.
- Use chatbot as an example
- Within the lifetime of a scope - be that a chat, an HTTP request, a socket connection or something else - you will have one application instance handling all the events from it, and you can persist things onto the application instance as well. You can choose to write a raw ASGI application if you wish, but Channels gives you an easy-to-use abstraction over them called *consumers*.
- 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.
- Routing Configuration is an ASGI application
- ASGI 异步服务器网关接口，其实用起来不是需要深入理解，先放一放
- You'll also need to point Channels at the root routing configuration. Edit the `mysite/settings.py` file again and add the following to the bottom of it:

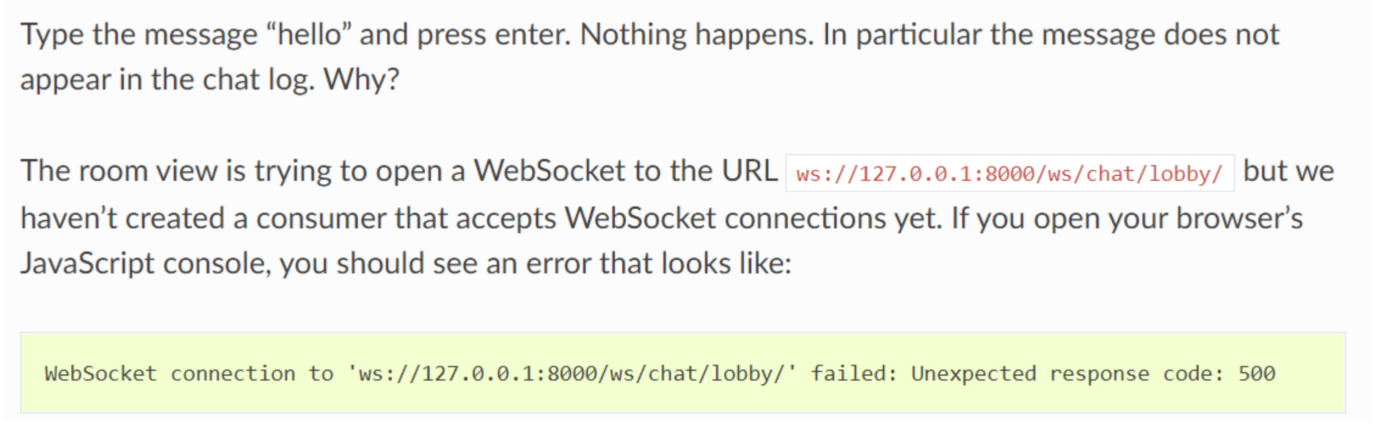
```
# mysite/settings.py
# Channels
ASGI_APPLICATION = 'mysite.asgi.application'
```
- Add ASGI configuration

- 

  You'll also need to point Channels at the root routing configuration. Edit the `mysite/settings.py` file again and add the following to the bottom of it:

  ```
  # mysite/settings.py
  # Channels
  ASGI_APPLICATION = 'mysite.asgi.application'
  ```

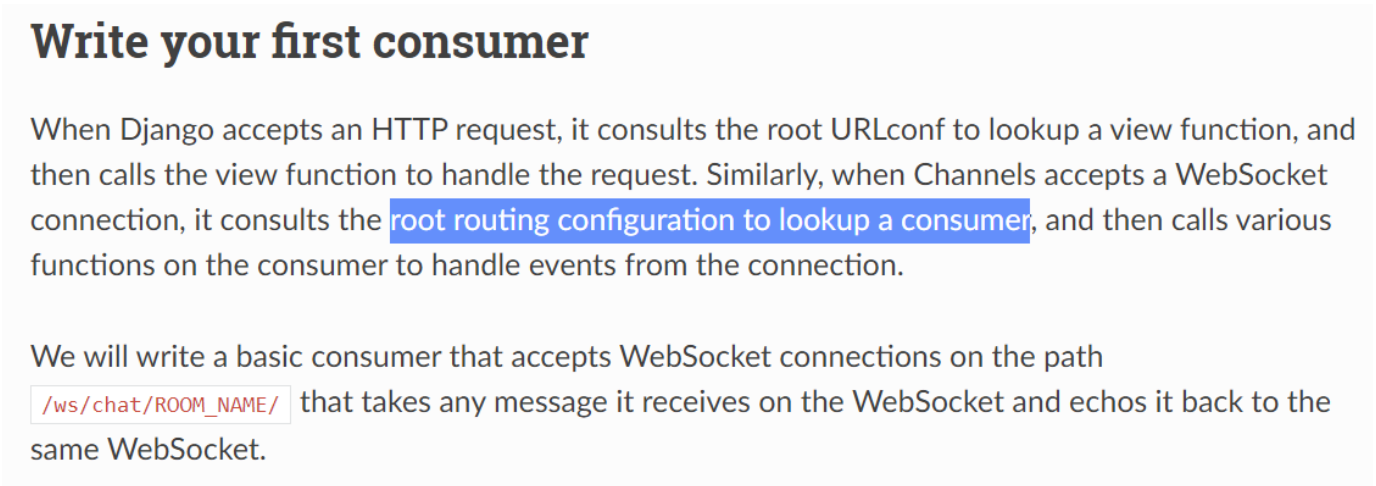
  要注意把mysite改成自己的那个名字

- 

  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:

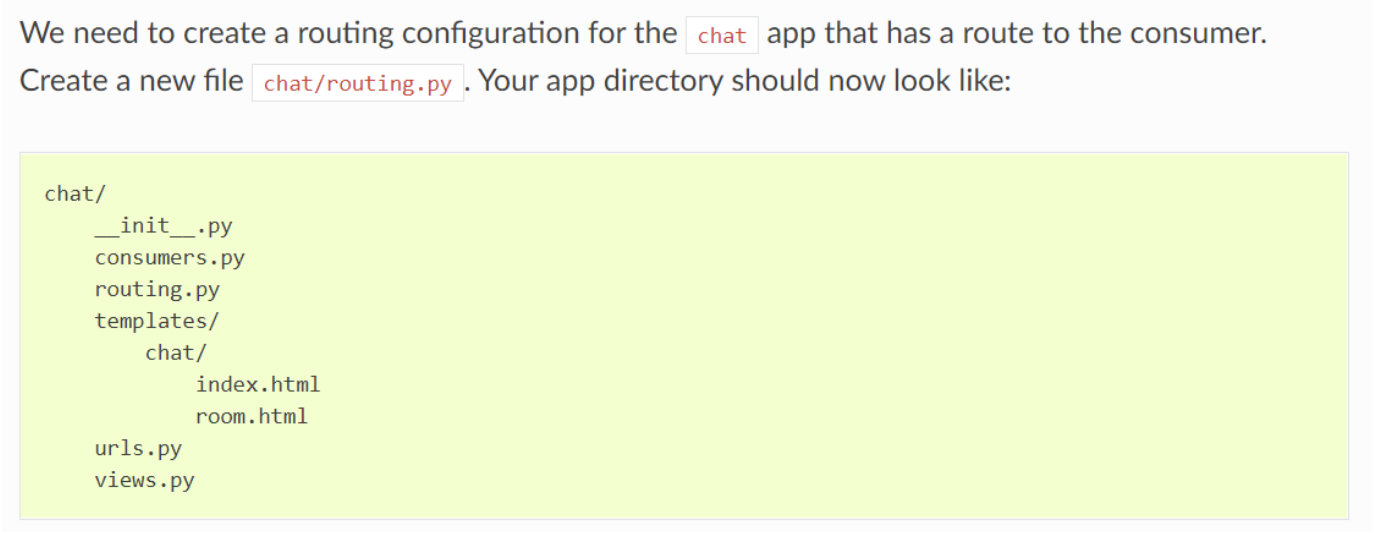
  ```
  WebSocket connection to 'ws://127.0.0.1:8000/ws/chat/lobby/' failed: Unexpected response code: 500
  ```

- 因为还没有创建consumer,因此还是消息显示不出来
- 

  ## 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.

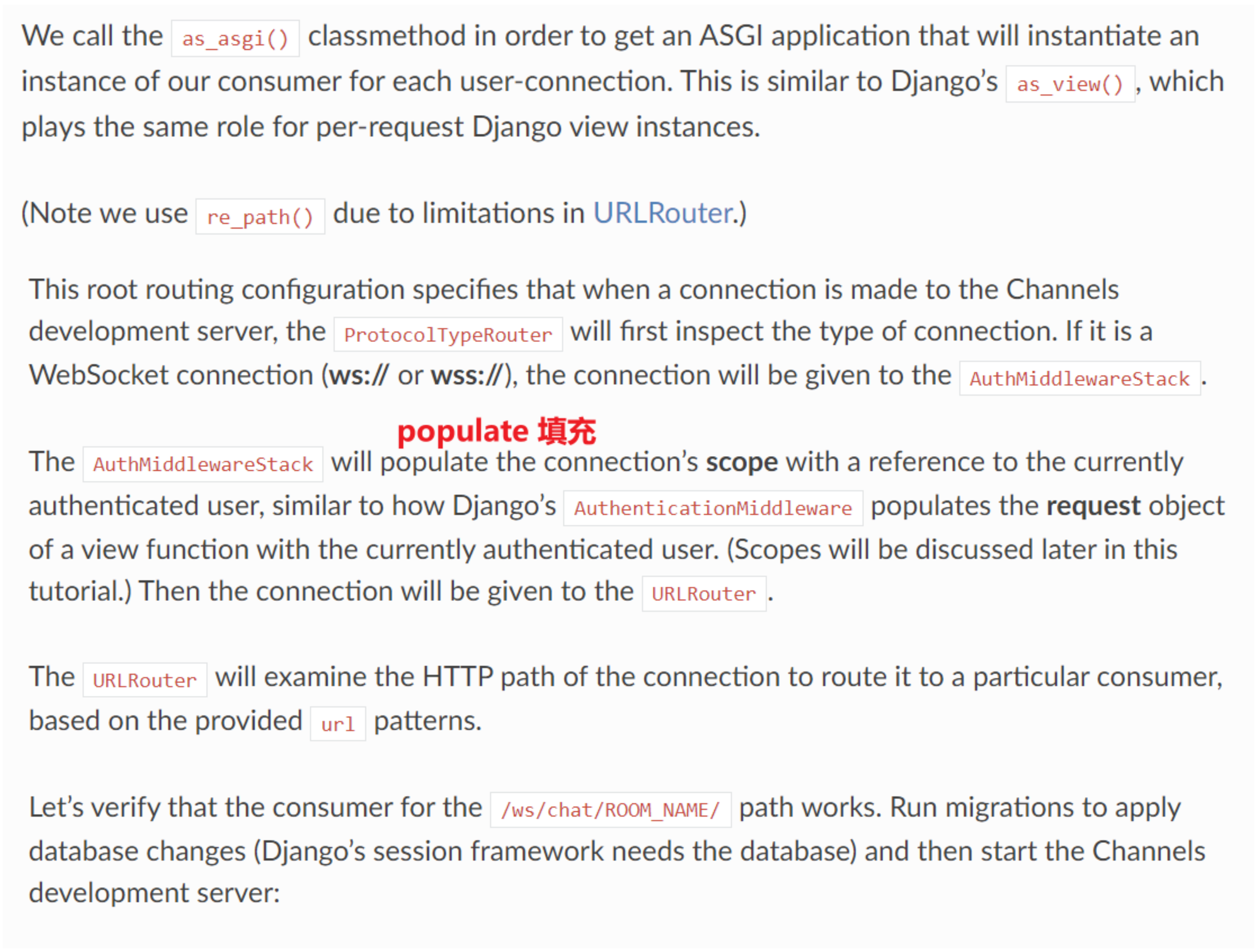
  We will write a basic consumer that accepts WebSocket connections on the path `/ws/chat/ROOM_NAME/` that takes any message it receives on the WebSocket and echos it back to the same WebSocket.

- 

  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
  ```

- Location of routing and templates
-


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.)

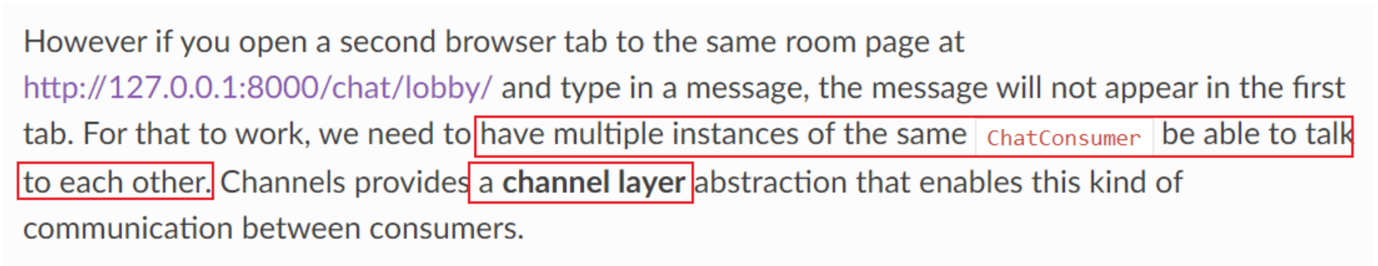
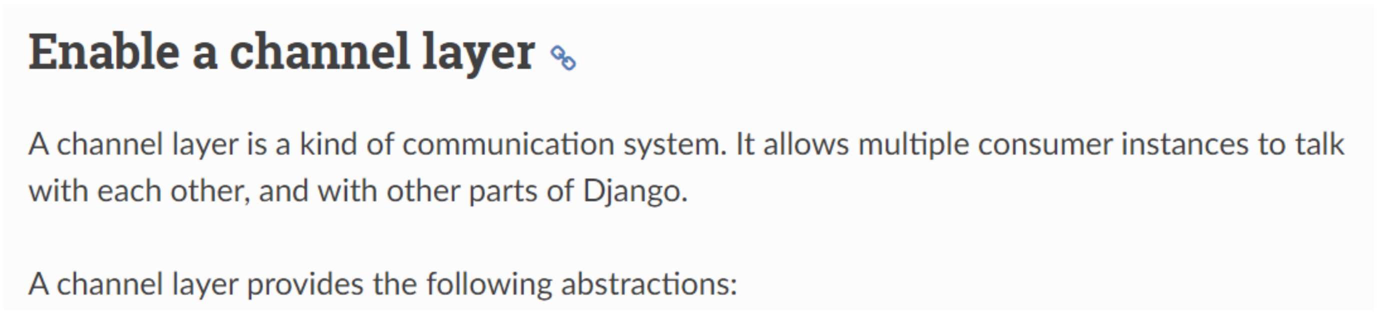
- This root routing configuration specifies that when a connection is made to the Channels development server, the `ProtocolTypeRouter` will first inspect the type of connection. If it is a WebSocket connection (**ws://** or **wss://**), the connection will be given to the `AuthMiddlewareStack`.

  populate 填充

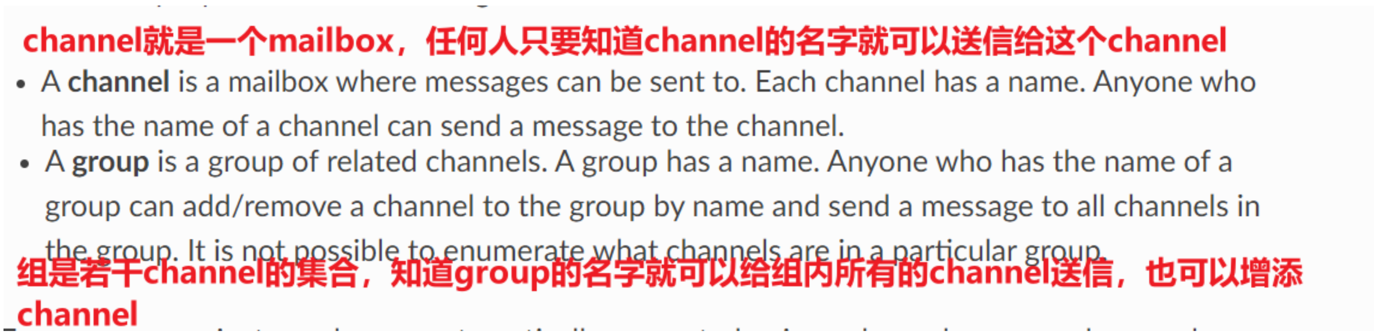
  The `AuthMiddlewareStack` will populate the connection's **scope** with a reference to the currently authenticated user, similar to how Django's `AuthenticationMiddleware` populates the **request** object of a view function with the currently authenticated user. (Scopes will be discussed later in this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

- 需要进行迁移
- 
  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.
- 想要在不同的browser同时看到消息的做法
- 
  ## 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:
- Channel layer提供了这种方法
- 
  channel就是一个mailbox，任何人只要知道channel的名字就可以送信给这个channel
  - 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.

  组是若干channel的集合，知道group的名字就可以给组内所有的channel送信，也可以增添channel

- Every consumer instance has an automatically generated unique channel name, and so can be communicated with via a channel layer.
- 每一个consumer都有一个自己独一无二的channel
- 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
  ```
- backing store 需要使用docker
- 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)],
          },
      },
  }
  ```
- 对应需要更新的设置
- 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.
- 上面是一个具体的流程
-

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 letters, digits, hyphens, and 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. 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.
    - `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.
- 对每个函数的各个部分的详细解释

```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
        }))
```

- 把同步改成更加强大的异步