django-channels-presence Documentation

Release 1.0.0

Charlie DeTar

Mar 30, 2020

Contents

`django-channels-presence` is a Django app which adds "rooms" and presence notification capability to a Django application using django-channels. If you're building a chat room or other site that needs to keep track of "who is connected right now", this might be useful to you.

CHAPTER 1

Quick install

1. Install with pip:

```
pip install django-channels-presence
```

2. Add "channels_presence" to INSTALLED_APPS:

```
# proj/settings.py

INSTALLED_APPS = [
    ...
    "channels_presence",
    ...
]
```

CHAPTER 2

Upgrading

Django Channels v2 is a major backwards-incompatible change from Django Channels v1; and hence the latest django-channels-presence is not compatible with older versions of django-channels.

If you're using:

- django-channels v1.x: Use django-channels-presence v0.
- django-channels v2.x: Use django-channels-presence v1+.

Code that uses `django-channels-presence` v0 or `django-channels` v1 will need to be rewritten to target channel layers, consumers, and other new concepts from `django-channels` v2.

CHAPTER 3

Motivation

This application builds on top of django-channels. You should have a good understanding of how it works before diving in here. To enable groups and messaging by channel name, `django-channels-presence` requires that the optional "channel layers" feature of `django-channels` v2 be used.

There are 3 main tasks that need to be accomplished in order to track "presence" in rooms using `django-channels`:

1. Observe connections, adding the channel names for each connecting socket to a group.
2. Observe disconnections, removing the channel names for each connecting socket from a group.
3. Prune channel names from groups after they go stale. We won't always get a socket disconnect event when a socket drops off; so we need to use heartbeats and a periodic pruning task to remove stale connections.

`django-channels-presence` provides database models and helpers to handle each of these tasks. This implementation makes database queries on every connection, disconnection, and message, as well as periodic queries to prune stale connections. As a result, it will scale differently than `django-channels` alone.

Contents:

3.1 Usage

3.1.1 Prerequisites

Install and set up django-channels and channel layers. A `CHANNEL_LAYERS` configuration is necessary to enable the use of consumer `channel_name` properties, to allow storing groups of channels by name.

3.1.2 Managing presences

In `django-channels-presence`, two main models track the presence of channels in a room:

- `Room`: represents a collection of channels that are in the same "room". It has a single property, `channel_name`, which is the "group name" for the channel layer group to which its members are added.

- `Presence`: represents an association of a single consumer channel name with a `Room`, as well as the associated auth `User` if any.

To keep track of presence, the following steps need to be taken:

1. Add channels to a `Room` when users successfully join.
2. Remove channels from the `Room` when users leave or disconnect.
3. Periodically update the `last_seen` timestamp for clients' `Presence`.
4. Prune stale `Presence` records that have old timestamps by running periodic tasks.
5. Listen for changes in presence to update application state or notify other users

3.1.2.1 1. Adding channels to rooms

Add a user to a `Room` using the manager `add` method. For example, this consumer adds the connecting user to a room on connection. This will trigger the channels_presence.signals.presence_changed signal:

```
from channels.generic.websocket import WebsocketConsumer
from channels_presence.models import Room

class MyConsumer(WebsocketConsumer):
    def connect(self):
        super().connect()
        Room.objects.add("some_room", self.channel_name, self.scope["user"])
```

Channel names could be added to a room at any stage – for example, in the `connect` handler, the `receive` handler, or wherever else makes sense. In addition to handling `Room` and `Presence` models, `Room.objects.add` takes care of adding the channel name to the named channel layer group.

3.1.2.2 2. Removing channels from rooms

Remove a consumer's channel from a Room using the manager `remove` method. For example, this handler for `disconnect` removes the user from the room on disconnect. This will trigger the `presence_changed` signal:

```
from channels.generic.websocket import WebsocketConsumer
from channels_presence.models import Room

class MyConsumer(WebsocketConsumer):
    def disconnect(self, close_code):
        Room.objects.remove("some_room", self.channel_name)
```

`Room.objects.remove` takes care of removing the specified channel name from the channels group.

For convenience, `channels_presence.decorators.remove_presence` is a decorator to accomplish the same thing:

```
from channels.generic.websocket import WebsocketConsumer
from channels_presence.decorators import remove_presence

class MyConsumer(WebsocketConsumer):
    @remove_presence
    def disconnect(self, close_code):
        pass
```

3.1.2.3 3. Updating the "last_seen" timestamp

In order to keep track of which sockets are actually still connected, we must regularly update the `last_seen` timestamp for all present connections, and periodically remove connections from rooms if they haven't been seen in a while.

```
from channels.generic.websocket import WebsocketConsumer
from channels_presence.models import Presence

class MyConsumer(WebsocketConsumer):
    def receive(self, close_code):
        Presence.objects.touch(self.channel_name)
```

For convenience, the `channels_presence.decorators.touch_presence` decorator accomplishes the same thing:

```
from channels.generic.websocket import WebsocketConsumer
from channels_presence.decorators import touch_presence

# handler for "websocket.receive"

class MyConsumer(WebsocketConsumer):
    @touch_presence
    def receive(self, text_data=None, bytes_data=None):
        ...
```

This will update the `last_seen` timestamp any time any message is received from the client. To ensure that the timestamp remains current, clients should send a periodic "heartbeat" message if they aren't otherwise sending data but should be considered to still be present.

3a. Heartbeats

To allow efficient updates, if a client sends a message which is just the JSON encoded string `"heartbeat"`, the `touch_presence` decorator will stop processing of the message after updating the timestamp. The decorator can be placed first in a decorator chain in order to stop processing of heartbeat messages prior to other costly steps.

If updating `last_seen` on every message is too costly, an alternative to using the `touch_presence` decorator is to manually call `Presence.objects.touch` whenever desired. For example, this updates `last_seen` only when the literal message `"heartbeat"` is received:

```
from channels.generic.websocket import WebsocketConsumer
from channels_presence.models import Presence

class MyConsumer(WebsocketConsumer):
    def receive(self, text_data=None, bytes_data=None):
        ...
        if text_data == '"heartbeat"':
            Presence.objects.touch(self.channel_name)
```

To ensure that an active connection is not marked as stale, clients should occasionally send `"heartbeat"` messages:

```
// client.js

setInterval(function() {
    socket.send(JSON.stringify("heartbeat"));
}, 30000);
```

The frequency should be adjusted to occur before the maximum age for last-seen presence, set with `settings.CHANNELS_PRESENCE_MAX_AGE` (default 60 seconds).

3.1.2.4 4. Pruning stale connections

In order to remove connections whose timestamps have expired, we need to periodically launch a cleaning task. This can be accomplished with `Room.objects.prune_presences()`. For convenience, this is implemented as a celery task which can be called with celery beat: `channels_presence.tasks.prune_presences`. The management command `./manage.py prune_presences` is also available for calling from cron.

A second maintenance command, `Room.objects.prune_rooms()`, removes any `Room` models that have no connections. This is also available as the celery task `channels_presence.tasks.prune_rooms` and management command `./manage.py prune_rooms`.

See the documentation for periodic tasks in celery details on configuring celery beat with Django. Here is one example:

```
# settings.py

CELERYBEAT_SCHEDULE = {
    'prune-presence': {
        'task': 'channels_presence.tasks.prune_presences',
        'schedule': timedelta(seconds=60)
    },
    'prune-rooms': {
        'task': 'channels_presence.tasks.prune_rooms',
        'schedule': timedelta(seconds=600)
    }
}
```

3.1.2.5 5. Listening for changes in presence

Use the `channels_presence.signals.presence_changed` signal to be notified when a user is added or removed from a Room. This is a useful place to define logic to update other connected clients with the list of present users. See the API reference for presence_changed for an example.

3.2 API Reference

3.2.1 Settings

`CHANNELS_PRESENCE_MAX_AGE` Default `60`. Maximum age in seconds before a presence is considered expired.

3.2.2 Models

3.2.2.1 `Room`

```
from channels_presence.models import Room
```

Manager:

`Room.objects.add(room_chanel_name, user_channel_name, user=None)` Add the given `user_channel_name` (e.g. `consumer.channel_name`) to a Room with the name `room_channel_name`. If provided, associate the auth `User` as well. Creates a new `Room` instance

if it doesn't exist; creates any needed `Presence` instance, and updates the channels group membership. Returns the `room` instance.

`Room.objects.remove(room_channel_name, user_channel_name)` Remove the given `user_channel_name` from the room with `room_channel_name`. Removes relevant `Presence` instances, and updates the channels group membership.

`Room.objects.prune_presences(age_in_seconds=None)` Remove any `Presence` models whose `last_seen` timestamp is older than `age_in_seconds` (defaults to `settings.CHANNELS_PRESENCE_MAX_AGE` if not specified).

`Room.objects.prune_rooms()` Remove any rooms that have no associated `Presence` models.

Instance properties:

`room.channel_name` The channel name associated with the group for this room.

Instance methods:

`room.get_users()` Return a queryset with all of the unique authenticated users who are present in this room.

`room.get_anonymous_count()` Return the number of non-authenticated sockets which are present in this room.

3.2.2.2 `Presence`

```
from channels_presence.models import Presence
```

Manager:

`Presence.objects.touch(channel_name)` Updates the `last_seen` timestamp to now for all instances with the given channel name.

`Presence.objects.leave_all(channel_name)` Removes all `Presence` instances with the given channel name. Triggers `channels_presence.signals.presence_changed` for any changed rooms.

Instance properties:

`presence.room` The room to which this Presence belongs

`presence.channel_name` The consumer channel name associated with this Presence

`presence.user` A `settings.AUTH_USER_MODEL` associated with this Presence, or None

`presence.last_seen` Timestamp for the last time socket traffic was seen for this presence.

3.2.3 Decorators

3.2.3.1 `touch_presence`

```
from chanels_presence.decorators import touch_presence
```

Decorator for use on `websocket.receive` handlers which updates the `last_seen` timestamp on any `Presence` instances associated with the client. If the message being sent is the literal JSON-encoded `"heartbeat"`, message processing stops and the decorator does not call the decorated function. Note that this decorator is syncronous, so should only be used on syncronous handlers.

```
from channels.generic.websocket import WebsocketConsumer

class MyConsumer(WebsocketConsumer):
    @touch_presence
    def receive(self, text_data=None, bytes_data=None):
        pass
```

3.2.3.2 remove_presence

```
from chanels_presence.decorators import remove_presence
```

Decorator for use on `websocket.disconnect` handlers which removes any `Presence` instances associated with the client. Note that this decorator is syncronous, so should only be used on syncronous handlers.

```
from channels.generic.websocket import WebsocketConsumer

class MyConsumer(WebsocketConsumer):
    @remove_presence
    def disconnect(self, close_code):
        pass
```

3.2.4 Signals

3.2.4.1 presence_changed

```
from channels_presence.signals import presence_changed
```

A Django signal dispatched on any addition or removal of a `Presence` from a `Room`. Use it to track when users come and go.

Arguments sent with this signal:

room The `Room` instance from which a `Presence` was added or removed.

added The `Presence` instance which was added, or `None`.

removed The `Presence` instance which was removed, or `None`.

bulk_change If `True`, indicates that this was a bulk change in presence. More than one presence may have been added or removed, and particular instances will not be provided in `added` or `removed` arguments.

Example:

```
# app/signals.py

import json

from asgiref.sync import async_to_sync
from channels.layers import get_channel_layer
from channels_presence.signals import presence_changed
from django.dispatch import receiver


channel_layer = get_channel_layer()
```

(continues on next page)

(continued from previous page)

```
@receiver(presence_changed)
def broadcast_presence(sender, room, **kwargs):
    """
    Broadcast the new list of present users to the room.
    """

    message = {
      "type": "presence",
      "payload": {
          "channel_name": room.channel_name,
          "members": [user.serialize() for user in room.get_users()],
          "lurkers": room.get_anonymous_count(),
      }
    }

    # Prepare a dict for use as a channel layer message. Here, we're using
    # the type "forward.message", which will magically dispatch to the
    # channel consumer as a call to the `forward_message` method.
    channel_layer_message = {
        "type": "forward.message",
        "message": json.dumps(message)
    }

    async_to_sync(channel_layer.group_send)(room.channel_name, channel_layer_message)
```

```
# app/channels.py: App consumer definition

from channels.generic.websocket import WebsocketConsumer

class AppConsumer(WebsocketConsumer):
    def forward_message(self, event):
        """
        Utility handler for messages to be broadcasted to groups.  Will be
        called from channel layer messages with `"type": "forward.message"`.
        """
        self.send(event["message"])
```

CHAPTER 4

Indices and tables

- genindex
- search