Django Chatter Documentation

Release 0.1.0

Ahmed Ishtiaque

Jun 02, 2019

Contents

Re-usable Django chat application for Django developers.

Chat is a crucial aspect of many web apps at present. However, Django's package repository does not have well-maintained reusable chat packages that Django developers can integrate into their platforms.

Django Chatter is an attempt to change that. This is an open-source fully reusable chat application that has mechanisms to support group chats in place.

The HTML front-end for this app is built with Flexbox, making it responsive to numerous viewports.

[More work to be done] Added to that, it can also possibly be used as a REST API, since all the views generate standard JSON responses that need to be parsed by the websockets present in the front-end of the app using this package.

This app makes use of Django Channels 2 and uses Redis as the message broker.

To run chatter properly, you'll require `python>=3.5` and Redis. **Note:** For development, we are currently using `redis-5.0.3`, built from source on Ubuntu machines.

The core mechanisms of Chatter follows the instructions provided in the Django Channels Tutorial section, with some added modifications and a little theming.

CHAPTER 1

Contents

1.1 Installation and Use

Installing Chatter requires several simple steps. While previous exposure to Django Channels would help you, it's not required.

1.1.1 Pre-requisites

Chatter only supports Python >= 3.5 and Django >= 2.0.9 as far as the developers are concerned. So, to be able to integrate Chatter, you're going to need them installed.

Added to that, Chatter uses Redis as its message broker. This means that all the chat messages are communicated between all connected users through the Redis datastore. Given that, you have to have Redis installed on your system. Details on installing Redis can be found on their Downloads page.

1.1.2 Installation

- Chatter is on PyPi now! To install it, run

```
pip install django-chatter
```

  This should install all the required dependencies for Chatter.

- Once you're done with that, add it to your `settings.INSTALLED_APPS`:

```
INSTALLED_APPS = [
  ...
  'django_chatter',
  ...
  ]
```

- Since we use Redis as our message broker, you need to enable channel layers for Chatter's ChatConsumer (see Channels' Consumers for more details). To enable that, you need to add the following lines to your project's `settings.py` file:

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

- If you haven't already, create a file named `routing.py` in your project's configuration folder. This is because Django Channels uses a specification called ASGI for its websocket protocol. To enable Channels on your app, you have to add a file that routes all websocket requests to a Channels app (in this case, Chatter). This should be the same as the folder where your `settings.py` file is located.

  In `routing.py`, add the following lines:

```
from channels.auth import AuthMiddlewareStack
from channels.routing import ProtocolTypeRouter, URLRouter
import django_chatter.routing

application = ProtocolTypeRouter({
  'websocket': AuthMiddlewareStack(
    URLRouter(
    django_chatter.routing.websocket_urlpatterns # send request to chatter's urls
    )
  )
})
```

  This routes all websocket requests to Chatter, with the logged in `User` object. If you are using different django-channels applications other than Chatter, you may already have this file, and can add the appropriate URL for chatter to handle. More details can be found on Django Channels' Routing page.

  If you know how the middleware wrapping in Channels works, then feel free to replace `AuthMiddlewareStack` with what you use as your auth middleware for User object processing (if you're curious to know about this, get in touch! We'd be happy to talk to you about it).

- Now that you're done setting up `routing.py`, add the following line in your `settings.py` file to enable routing websocket requests to the appropriate app:

```
ASGI_APPLICATION = 'mysite.routing.application'
```

- Link `django_chatter.urls` to the URL you want in your URLConf (`<project>/urls.py`).

  Example:

```
from django.urls import path, include

...
urlpatterns = [
  ...,
  path('chat/', include('django_chatter.urls')),
  ...
]
```

- Run migrations:

```
$ python manage.py makeimigrations django_chatter
$ python manage.py migrate
```

- Start your app's development server and go to your `'/chat/'` URL, and you will see Chatter's homepage.

1.1.3 Usage Notes

- Chatter, as of right now, provides a very minimal interface for users to chat with other users.For starters, while group chatting is supported on the model layer, the corresponding templates and front-end logic have not yet been setup.
- If you're using chatter as a package in your own app, you have to make sure that you handle user authentication in your app. Chatter, by default, provides views that require user authentication. If you're developing Chatter on the other hand, the usage will vary a bit. The notes for that can be found in the *Get Involved* section.

1.2 Example(s)

An example video of Chatter in action will be posted here very soon. Thanks for waiting!

1.3 Customizing Django Chatter Templates

- **Host Chatter Templates Inside Your App**

  You can add chatter templates inside your own app's HTML files. For example, if you have reusable headers and footers in a `base.html` file, then you can include that file's location in your `settings` file like so:

```
CHATTER_BASE_TEMPLATE="<your app templates directory>/base.html"
```

  Depending on how your template directories are defined, Django will try to find the template located in the location you've defined, and use it as a container for Chatter.

1.4 Utilities

1.4.1 Available

Chatter has the following utilities available:

- **Middleware to Support Multitenancy**

  Added in: Chatter 0.1.0

  Django Chatter now supports multitenant SaaS applications made using django-tenants. This is made available as middlewares in the `utils.py` module. Both these middlwares require *CookieMiddleware* and *SessionMiddleware* stacked higher in the ASGI application routing stack.

  - *MTSchemaMiddleware*:

    This middleware attaches `schema_name` as well as a boolean named `multitenant` into a websocket consumer's `scope`. This enables you to access the schema name from any consumer that's wrapped inside this middleware. To do this, you have to add it into your middleware stack in your project's *routing.py* file like so:

```
from django_chatter.utils import MTSchemaMiddleware

application = ProtocolTypeRouter({
 'websocket': <your stack>(
    MTSchemaMiddleware(
    URLRouter(
     django_chatter.routing.websocket_urlpatterns
     )
    )
    )
})
```

After doing this, your consumers will have access to the schema_name that you can use with *django-tenant*'s *schema_context*.

– *MTAuthMiddleware*:

This middleware is Chatter's version of attaching a user object to a websocket's `scope`. This automatically attaches the logged in user's information from the client's session cookies depending on which tenant they're accessing Chatter from. You can use it in your project's `routing.py` by the following method:

```
from django_chatter.utils import MTAuthMiddleware

application = ProtocolTypeRouter({
 'websocket': <your stack>(
    MTAuthMiddleware(
    URLRouter(
     django_chatter.routing.websocket_urlpatterns
     )
    )
    )
})
```

There's a high chance that you'd want to be using both these middlewares. To make things easy, these two are combined with `CookieMiddleware` and `SessionMiddleware` to make `ChatterMTMiddlewareStack` which you can use like this:

```
from django_chatter.utils import ChatterMTMiddlewareStack

application = ProtocolTypeRouter({
 'websocket': <your stack>(
    ChatterMTMiddlewareStack(
    URLRouter(
     django_chatter.routing.websocket_urlpatterns
     )
    )
    )
})
```

- **Create Room Function**

Added in: Chatter 0.1.1

This function takes in a _list_ of User objects and returns the ID of a new room containing the given users. If the room already exists in the database, it returns the existing room's ID. With the ID, you can then call upon Chatter's `chatroom` view from your view. An example is below:

```
from django_chatter.utils import create_room
from django_chatter.views import chatroom
from myapp.models import User

def my_view(request):
  user1 = request.user  # User requesting the view
  user2 = User.objects.get(username="user2")  # example user in your db
  room_id = create_room([user1, user2])
  return chatroom(request, room_id)
```

The above code would create a room from your view, and direct the user to the newly formed room.

1.4.2 To Do

Some utilities would be nice to have integrated with Chatter. For example, we could have the following:

- A module that takes in a list of `User` objects and creates a room with them in it, and returns the UUID of the new Room.

1.5 Tests

This project comes with unit tests to make sure the code is reliable. We welcome test contributions from developers. Especially, the multitenancy support testing is still in progress, so that needs a lot of work.

1.6 Get Involved with Django Chatter!

We'd highly appreciate contributions to this project! The source code is currently hosted on GitHub. If you caught a bug or have suggestions, we welcome pull requests and issues!

Currently, Ahmed Ishtiaque is the primary maintainer for this project. If you have any questions or suggestions, feel free to reach out to him as well.

1.6.1 Get Started

To start developing Chatter, follow the following steps:

- Install Python 3 if you don't have it already.
- Clone the GitHub Repo.
- Spawn up a new virtual environment, `cd` into your working directory and run

```
$ pip install -r dev-requirements.txt
```

This will install all the prerequisites needed to run Chatter.

- If you don't have Redis, you can install it from their Download page.
- Since we're phasing into implementing multitenancy support on Chatter with django-tenants, we will be using PostgreSQL as the database. Install PostgreSQL from PostgreSQL.

  After this, create user for chatter database:

  - Open the postgres terminal:

```
$ sudo su - postgres
```

- Connect to your psql server:

```
$ psql
```

- Run the following commands (don't miss the semi-colons):

```
$ CREATE DATABASE chatter;
$ CREATE USER chatteradmin WITH PASSWORD 'chatter';
$ ALTER ROLE chatteradmin SET client_encoding TO 'utf8';
$ ALTER ROLE chatteradmin SET default_transaction_isolation TO 'read committed
↪';
$ ALTER ROLE chatteradmin SET timezone TO 'America/New_York';
$ GRANT ALL PRIVILEGES ON DATABASE chatter TO chatteradmin;
$ \q
```

The instructions should be pretty intuitive. This is a replication of the detailed PostgreSQL install guide on DigitalOcean.

- Exit the postgres session:

```
$ exit
```

- Run migrations:

```
$ python manage.py makemigrations django_chatter
$ python manage.py migrate
```

- Create public tenant to enable multitenancy testing support with django-tenants:

```
$ python manage.py shell
```

```
from tenants.models import Client, Domain

# create your public tenant
tenant = Client(schema_name='public',
                name='Schemas Inc.')
tenant.save()

# Add one or more domains for the tenant
domain = Domain()
domain.domain = 'localhost' # don't add your port or www here! on a local server␣
↪you'll want to use localhost here
domain.tenant = tenant
domain.is_primary = True
domain.save()
```

- Run the tests:

```
$ pytest
```

All tests in the `master` branch should pass.

- Create a superuser for chatter:

```
$ python manage.py createsuperuser
```

- Run the development server:

```
$ python manage.py runserver
```

- (Optional) if you want to streamline the login/logout mechanisms, feel free to add a `login.html` file to *django_chatter/templates/registration* folder. This should give you a form to log in. Django's template for that is pretty adequate.

The following is a list of features and hooks that we plan on bringing to Chatter:

1.6.2 Features Yet to Come

- Add a "Create Group" option for users on the templates
- Add 'Seen by user x' functionality
- Add time to when messages were sent

1.7 Credits

We would like to thank Andrew Godwin, along with the Django Software Foundation, for providing the wonderful and easy-to-use Django Channels framework to base our websocket-based app on.

1.8 Changelog

1.8.1 v 1.0.7

- Bugfix: Properly selecting the last 10 rooms when loading a chat window.

1.8.2 v 1.0.6

- UI improvement: the Opponent username bubble has alignment flex-end to ensure it's in the bottom
- Cache usernames of all users present in a room to save database queries

1.8.3 v 1.0.5

- Major change: Now, whenever a user/client connects to a room, the UI gets updated if they receive messages in a separate room. This is achieved by connecting to an additional websocket that is defined by the user's username. New alerts are received in this websocket and the update is added to the UI.
- Tests are added to test this new websocket consumer's behavior.
- Refactored more of the JS code into their own files and added the dependencies on the top of each file.
- Minor UI improvements to keep things intuitive

1.8.4 v 1.0.4

- Bugfixes: The last message preview on chatroom-list updates as the websocket receives new messages. Overflow of text in the preview has been adjusted for.

1.8.5 v 1.0.3

- Minor bugfix: Use relative URL when fetching messages to account for parent app's URL settings.

1.8.6 v 1.0.2

- On click, messages show when they were sent.
- Fresh UI, inspired by Google's Messages Web and Facebook Messenger.
- ChatConsumer now sends and receives JSON data by default.
- More modern dropdown for selecting users. This has been put in place to allow group chat formation in the future.
- Infinite scroll to retrieve previous messages has been implemented.

1.8.7 v 1.0.1

- Cleaned up some testing code
- Bugfix in MTSchemaMiddleware - hostname to search tenant with was only the first part of the domain instead of the whole domain.

1.8.8 v 1.0.0

- This version removes the context processor `get_chatroom_list` that used to fetch a list of all rooms a logged in user is a member of. This is to prevent unnecessary database access in the request-response cycle. For users using django_chatter < 1.0.0, this will create compatibility issues, which can be solved by simply removing the context processor from their settings.
- Multiple tests have been added to maintain reliability of the code.
- On multitenant systems, `MTSchemaMiddleware` checks if a tenant with the given schema name exists. If not, it raises an Http404 error.

1.8.9 v 0.2.2

- Added testing framework for multitenancy support
- Switched to class-based views to promote clearer code style
- index page bugfix
- Added coverage and Travis CI information