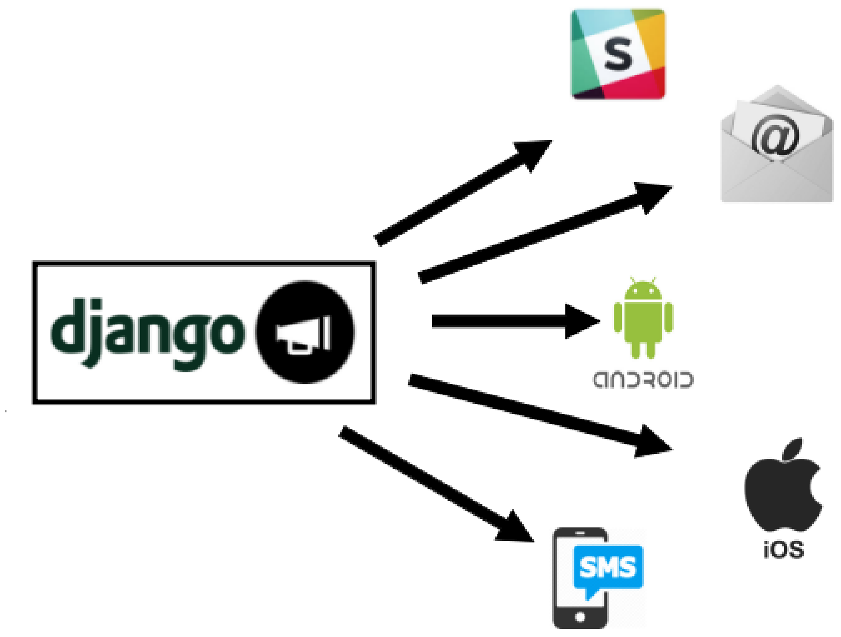
django-notifs

Daniel Osaetin

Nov 13, 2021

CONTENTS



django-notifs is a modular notifications app for Django that basically allows you to notify users about events that occur in your application E.g

- Your profile has been verified
- User xxxx sent you a message

It also allows you to deliver these notifications to any destination you want to with custom delivery channels.

It also supports asynchronous notification with several pluggable delivery backends (e.g Celery, RQ etc)

CHAPTER ONE

EXAMPLES?

A tutorial on how to build a Realtime Chat application with Vue, django-notifs, RabbitMQ and uWSGI

The Repository for the chat app (Chatire) is also available on github

CHAPTER TWO

DOCUMENTATION

https://django-notifs.readthedocs.io

CHAPTER

THREE

CONTENTS

3.1 Overview

3.1.1 Requirements

- Python 3.6+
- Django 2.2+

3.1.2 Supported Functionality

- In-app notifications
- Silent notifications (i.e Notifications that aren't saved in the database)
- Delivery providers e.g Email, Slack, SMS etc
- Custom delivery channels and providers
- Asynchronous notifications (with support for multiple backends e.g Celery, RQ, AwsLambda etc)

3.1.3 Supported providers

- *Email*: SMTP and Transaction email providers (Amazon SES, Mailgun, Mailjet, Postmark, SendinBlue, SendGrid, SparkPost and Mandrill) with django-anymail
- *SMS*: (Twilio, Messagebird) with django-sms
- *Slack*
- *Pusher Channels*
- *Twitter (Status updates)*
- *FCM - Web push notifications* with pyFCM
- *Django channels*

3.2 Installation

Get it from pip with:

```
pip install django-notifs
```

Include it in `settings.INSTALLED_APPS`:

```
INSTALLED_APPS = (
    'django.contrib.auth',
    ...
    'notifications',
    'django_jsonfield_backport'  # if you're running django < 3.1
    ...
)
```

Finally don't forget to run the migrations with:

```
python manage.py migrate notifications
```

You can also register the current Notification model in django admin:

```
"""admin.py file."""
from django.contrib import admin
from notifications.utils import get_notification_model


Notification = get_notification_model()
admin.site.register(Notification)
```

3.3 Usage

3.3.1 Quick start

To Create/Send a notification import the notify function and call it with the following arguments:

```
from notifications.utils import notify

notify(
    **notification_kwargs,  # Notification kwargs that map to the current␣
↪Notification model
    silent=True,  # Don't persist to the database
    countdown=0  # delay (in seconds) before sending the notification
    channels=('email', 'slack'),
    extra_data={
        'context': {}  # Context for the specified Notification channels
    }
)
```

This example creates a *silent* notification and delivers it via `email` and `slack`.

This assumes that you've implemented these channels

A *NotificationChannel* is a class thats builds a payload from a Notification object and sends it to one or more providers. Below is an example of a channel that builds a payload containing the context and provider and then, delivers it to the inbuilt Console provider (which simply prints out any payload that it receives):

```python
from notifications.channels import BaseNotificationChannel


class CustomNotificationChannel(BaseNotificationChannel):
    name = 'custom_notification_channel'
    providers = ['email']

    def build_payload(self, provider):
        return {'context': self.context, 'payload': provider}
```

Note: The `build_payload` method accepts the current provider as an argument so you can return a different payload based on the current provider.

Then you can instantiate the Notification channel directly:

```python
console_notification = ConsoleNotificationChannel(
    notification: Notification, context={'arbitrary_data': 'data'}
)
console_notification.notify()  # Send immediately
console_notification.notify(countdown=60)  # Send after 1 minute
```

This gives you more flexibility over the `notify` utility function because you can create several notifications and decide how each individual notification should be sent

Note: Notification channels are automatically registered by django-notifs You must inherit from the base class and specify the `name` property for the channel to be properly registered

3.3.2 Bulk sending

You can send bulk notifications by setting the `bulk` property to `True` in the context dictionary:

```python
console_notification = ConsoleNotificationChannel(
    notification: Notification, context={'bulk': True, 'arbitrary_data': 'data'}
)
console_notification.notify()
```

or:

```python
notify(
    ...,
    extra_data={
        'context': {
            'bulk': True,
        }
    }
)
```

Note: The provider takes care of sending the payload in the most efficient way. (Some providers like `pusher_channels` have a bulk api for delivering multiple notifications in a single batch).

3.3.3 Notification Model

Django notifs includes an inbuilt notification model with the following fields:

- **source: A ForeignKey to Django's User model (optional if it's not a User to User Notification).**
- **source_display_name: A User Friendly name for the source of the notification.**
- **recipient: The Recipient of the notification. It's a ForeignKey to Django's User model.**
- **category: Arbitrary category that can be used to group messages.**
- **action: Verbal action for the notification E.g Sent, Cancelled, Bought e.t.c**
- **obj: An arbitrary object associated with the notification using the `contenttypes` app (optional).**
- **short_description: The body of the notification.**
- **url: The url of the object associated with the notification (optional).**
- **silent: If this Value is set, the notification won't be persisted to the database.**
- **extra_data: Arbitrary data as in a JSONField.**
- **channels: Notification channels related to the notification (Tuple/List in a JSONField)**

The values of the fields can easily be used to construct the notification message.

3.3.4 Extra/Arbitrary Data

Besides the standard fields, django-notifs allows you to attach arbitrary data (as JSON) to a notification. Simply pass in a dictionary as the extra_data argument.

Note: This field is only persisted to the database if you use use the default Notification model or a custom model that provides an `extra_data` field.

3.3.5 Sending notifications asynchronously

`django-notifs` is designed to support different backends for delivering notifications. By default it uses the `Synchronous` backend which delivers notifications synchronously.

Note: The Synchronous backend is not suitable for production because it blocks the request. It's more suitable for testing and debugging. To deliver notification asynchronously, please see the *backends section*.

3.3.6 Delayed notifications

You can delay a notification by passing the `countdown` (in seconds) parameter to the `notify` function:

```
# delay notification for one minute
notify(**kwargs, countdown=60)
```

3.3.7 Reading notifications

To read a notification use the read method:

```python
from notifications.utils import read

# id of the notification object, you can easily pass this through a URL
notify_id = request.GET.get('notify_id')

# Read notification
if notify_id:
    read(notify_id=notify_id, recipient=request.user)
```

Note: It's really important to pass the correct recipient to the `read` function.

Internally,it's used to check if the user has the right to read the notification. If you pass in the wrong recipient or you omit it entirely, `django-notifs` will raise a `NotificationError`

3.3.8 Signals

pre_send

This signal is sent before a notification is sent

Argument	Value
`sender`	The provider class
`context`	The Notification context
`payload`	The Notification payload

post_send

This signal is sent after a notification is sent

Argument	Value
`sender`	The provider class
`context`	The Notification context
`payload`	The Notification payload
`response`	The Response from the provider

pre_bulk_send

This signal is sent before a bulk notification is sent

Argument	Value
`sender`	The provider class
`context`	The Notification context
`payload`	The Notification payload

post_bulk_send

This signal is sent after a bulk notification is sent

Argument	Value
`sender`	The provider class
`context`	The Notification context
`payload`	The Notification payload
`response`	The Response from the provider

3.4 Configuration

3.4.1 `NOTIFICATIONS_MODEL`

`Default='notifications.models.Notification'`

This setting is used override the default database Model for saving notifications. Most users wouldn't need to override this but it can be useful if you're trying to integrate django-notifs into an existing project that already has it's own Notificaiton model

3.4.2 `NOTIFICATIONS_DELIVERY_BACKEND`

`Default='notifications.backends.Synchronous`

`django-notifs` is designed to support different backends for delivering notifications. By default it uses the `Synchronous` backend which delivers notifications synchronously.

Note: The Synchronous backend is not suitable for production because it blocks the request. It's more suitable for testing and debugging. To deliver notification asynchronously, please see the *backends section*.

3.4.3 `NOTIFICATIONS_QUEUE_NAME`

`Default='django_notifs'`

This setting is only valid for the Celery, Channels and RQ backend

This is the queue name for backends that have a "queue" functionality

3.4.4 `NOTIFICATIONS_RETRY`

`Default=False`

Enable the retry functionality.

The Retry functionality is only valid for the Celery and RQ backends

NOTIFICATIONS_RETRY_INTERVAL

`Default=5`

The retry interval (in seconds) between each retry

NOTIFICATIONS_MAX_RETRIES

`Default=5`

The maximum number of retries for a notification

3.5 Backends

The primary function of **a delivery backend** is to execute the code of the delivery channels and providers. *Unlike notification channels, you can only use one delivery backend at a time.*

3.5.1 Celery

Install the optional Celery dependency with:

```
pip install django-notifs[celery]
```

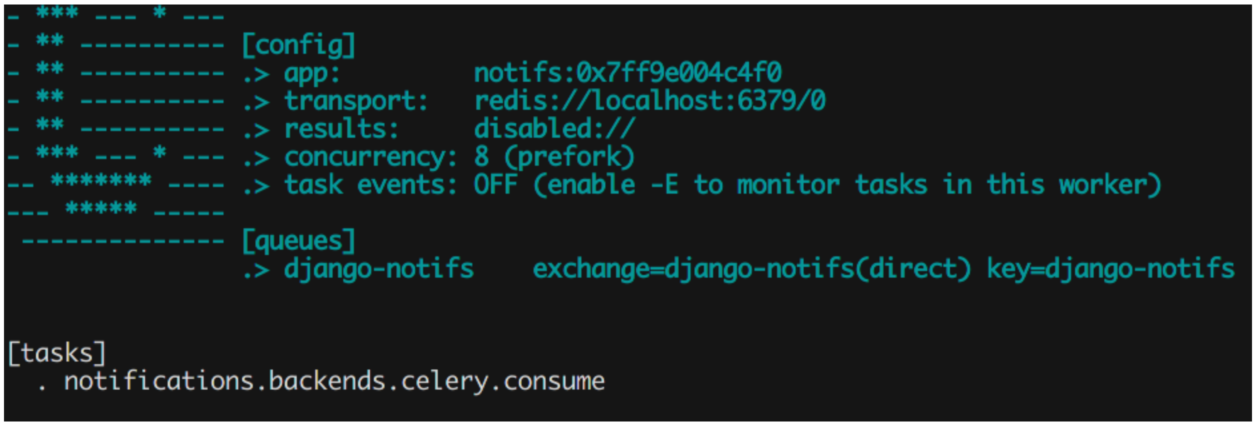
Enable it by setting `NOTIFICATIONS_DELIVERY_BACKEND` to `notifications.backends.Celery`

Run celery with the command:

```
celery -A yourapp worker -l info -Q django-notifs
```

Whenever a notification is created, it's automatically sent to celery and processed.

Make sure you see the queue and task (`notifications.backends.celery.consume`) in the terminal.



```
- *** --- * ---
- ** ---------- [config]
- ** ---------- .> app:         notifs:0x7ff9e004c4f0
- ** ---------- .> transport:   redis://localhost:6379/0
- ** ---------- .> results:     disabled://
- *** --- * --- .> concurrency: 8 (prefork)
-- ******* ---- .> task events: OFF (enable -E to monitor tasks in this worker)
--- ***** -----
 -------------- [queues]
                .> django-notifs    exchange=django-notifs(direct) key=django-notifs


[tasks]
  . notifications.backends.celery.consume
```

If you have issues registering the task, you can import it manually or checkout the Celery settings in the repo.

3.5.2 Channels

Install the channels dependency with:

```
pip install django-notifs[channels]
```

This also installs channels_redis as an extra dependency

Declare the notifications consumer in `asgi.py`:

```
from notifications import consumers

application = ProtocolTypeRouter({
    ...,
    'channel': ChannelNameRouter({
        'django_notifs': consumers.DjangoNotifsConsumer.as_asgi(),
    })
})
```

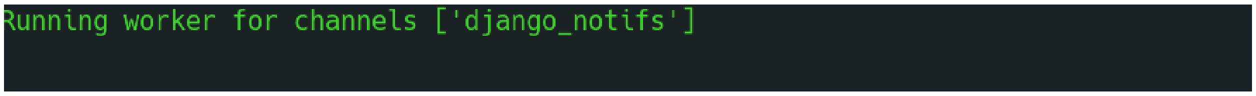
This example assumes that you're running Django 3x Which has native support for asgi. Check the channels documentation for Django 2.2

Next add the *django_notifs* channel layer to `settings.CHANNEL_LAYERS`:

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

Finally, run the worker with:

```
python manage.py runworker django_notifs
```



```
Running worker for channels ['django_notifs']
```

3.5.3 RQ

RQ is a lightweight alternative to Celery. To use the RQ Backend, install the optional dependency with:

```
pip install django-notifs[rq]
```

django notifs uses django-rq under the hood

Enable it by setting `NOTIFICATIONS_DELIVERY_BACKEND` to `notifications.backends.RQ`

Configure the `django_notifs` in `settings.py`:

```
RQ_QUEUES = {
    ...,
    'django_notifs': {
        'HOST': 'localhost',
```

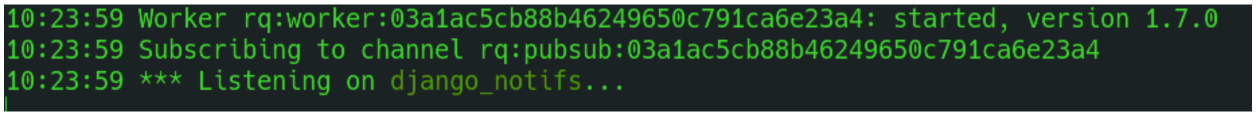
(continues on next page)

(continued from previous page)

```
        'PORT': 6379,
        'DB': 0,
        'PASSWORD': '',
        'DEFAULT_TIMEOUT': 360,
    }
}
```

Finally start the rq worker with:

```
python manage.py rqworker django_notifs --with-scheduler
```



```
10:23:59 Worker rq:worker:03a1ac5cb88b46249650c791ca6e23a4: started, version 1.7.0
10:23:59 Subscribing to channel rq:pubsub:03a1ac5cb88b46249650c791ca6e23a4
10:23:59 *** Listening on django_notifs...
```

See the django-rq documentation for more details

3.5.4 AwsLambda (with SQS)

The setup for this backend is more involved but it's probably the cheapest and most scalable backend to use in production because the heavylifting and execution environment is handled by AWS.

set `NOTIFICATIONS_DELIVERY_BACKEND` to `notifications.backends.AwsSqsLambda`

This backend uses `boto3` under the hood; so make sure your AWS credentials are configured e.g:

```
export AWS_ACCESS_KEY_ID=xxxx
export AWS_SECRET_ACCESS_KEY=xxxx
export AWS_DEFAULT_REGION=xxxx
```

Clone the lambda worker repository and run:

```
npm install
```

The `sqs-lambda-worker` folder includes four files that are of interest:

`.env.example`

You can use this file (after renaming it to `.env`) to configure the environment variables for the autogenerated Lambda function. You can replace this step by:

- Configuring the environment variables in your CI/CD environment **(Recommended)**
- Exporting them in the current shell.

This is useful if you want to test the serverless deployment locally before moving it to your CI/CD

`requirements.txt`

In order to keep the lambda function as lean as possible, you have to explicitly declare the requirements that are necessary for the lambda function. New providers (and their dependencies) are continuously added to django-notifs so it's not adviseable to install dependencies for providers that you don't need because this could impact the startup time of your Lambda function.

`serverless.yml`

The Serverless file. It contains a blueprint that deploys the simplest configuration possible but the configuration options are endless. see the Serverless documentation for AWS for more information.

`settings.py`

Declare the Django settings for the lambda function.

After setting these variables deploy the serverless stack to AWS:

```
serverless deploy --stage <your-stage>
```

Then update your settings with the generated sqs queue url:

```
settings.NOTIFICATIONS_SQS_QUEUE_URL = 'xxxxxx'    # autogenerated SQS url
```

3.5.5 Synchronous

This is the default backend that sends notifications synchronously.

You can enable it explicitly by setting `NOTIFICATIONS_DELIVERY_BACKEND` to `notifications.backends.Synchronous`

3.6 Providers

Django notifs comes with a set of inbuilt providers. These providers are typically classes that accept a payload and contain the logic for delivering the payload to an external service.

Below are the list of supported providers:

3.6.1 Email

name: `'email'`

The email provider uses the standard `django.core.mail` module. This opens up support for multiple ESP's (Mailjet, Mailchimp, sendgrid etc)

Installation

Optional dependency for django-anymail:

```
pip install django-notifs[anymail]
```

Settings

If you use `django-anymail` or a custom Email backend, all you have to do configure the settings and dependencies as you'd normally do and the email provider should pick it up.

Payload

You can still pass extra keyword arguments like `tags` (*depending on the ESP that you use.*) See the `django-anymail` documentation for more information.

3.6.2 SMS (with django-sms)

name: `'django_sms'`

The SMS provider uses a third-party app called django-sms this also opens up support for multiple SMS providers.

Supported providers are:

- Twilio
- Message bird

Installation

```
pip install django-notifs[django_sms]
```

Extra dependencies can be installed by:

```
pip install django-sms[twilio,messagebird]
```

Settings

See the django-sms documentation for more information on how to configure your preferred backend. Once it is configured, `django-notifs` should pick it up

Payload

3.6.3 Slack

name: `'slack'`

Installation

```
pip install django-notifs[slack]
```

Settings

```
NOTIFICATIONS_SLACK_BOT_TOKEN=xxxxxxx
```

Payload

3.6.4 Pusher Channels

name: `'pusher_channels'`

Installation

```
pip install django-notifs[pusher_channels]
```

Settings

```
NOTIFICATIONS_PUSHER_CHANNELS_URL=https://<app_id>:<app_secret>@api-eu.pusher.com/
↪apps/0000000
```

Payload

3.6.5 FCM (Firebase Web push)

name: `'fcm_web'`

Settings

```
NOTIFICATIONS_FCM_WEB_API_KEY=xxxxxxx  # FCM Api key
NOTIFICATIONS_FCM_WEB_PROXY = {}  # FCM proxy
```

Payload

Single:

Bulk:

3.6.6 Twitter status update

django-notifs uses tweepy to deliver twitter notifiations

name: `'twitter_status_update'`

Installation

```
pip install django-notifs[twitter]
```

Settings

NOTIFICATIONS_TWITTER_CONSUMER_KEY

Twitter consumer key

NOTIFICATIONS_TWITTER_CONSUMER_SECRET

Twitter consumer secret

NOTIFICATIONS_TWITTER_ACCESS_TOKEN

Twitter access token

NOTIFICATIONS_TWITTER_ACCESS_TOKEN_SECRET

Twitter access token secret

Payload

See the tweepy documentation for more information on these parameters

3.6.7 django-channels

name: `'django_channels'`

Installation

```
pip install django-notifs[channels]
```

Settings

`NOTIFICATIONS_WEBSOCKET_EVENT_NAME`

`Default='notifs_websocket_message'`

The `type` value of the messages that are going to received by the django notifs websocket consumer. In most cases, you don't need to change this setting.

`NOTIFICATIONS_WEBSOCKET_URL_PARAM`

`Default = 'room_name'`

The WebSocket URL param name. It's also used to construct the WebSocket URL. See the *Advanced usage* section for more information.

Context

```
{
    'channel_layer': "Custom django channels layer or 'default'",
    'destination': 'Group/channel name'
}
```

Payload

3.6.8 Writing custom Providers

Sometimes, the inbuilt providers are not sufficient to handle every use case.

You can create a custom provider by inheriting from the Base provider class or an existing Provider and Implementing the `get_validator/validate`, `send` and `send_bulk` method.

The Notification context is also available as a property (`self.context`):

```
from typing import Dict, List

from pydantic import BaseModel

from notifications.providers import BaseNotificationProvider
```

(continues on next page)

(continued from previous page)

```python
class CustomProviderSchema(BaseModel):
    event: str
    message: Dict


class BulkCustomProviderSchema(BaseModel):
    group: str
    messages: List[CustomProviderSchema]


class CustomNotificationProvider(BaseNotificationProvider):
    name = 'custom_provider'
    validator = CustomProviderSchema

    def get_validator(self):
        """Return a custom validator based on the context."""
        if self.context.get('bulk', False) is True:
            return BulkCustomProviderSchema

        return CustomProviderSchema

    def validate(self, payload):
        """Validate without pydantic."""
        pass

    def send(self, payload):
        # call an external API?
        pass

    def send_bulk(self, payloads):
        for payload in payloads:
            self.send(payload)

        # or call an external bulk API?
```

3.7 Advanced usage

3.7.1 Tentative Notifications

A tentative notification is a conditional notification that should only be sent if a criteria is met.

An example is sending a notification if a user hasn't read a chat message in 30 minutes (as a reminder).

You can acheive this by combining the `countdown` functionality with a custom provider:

```python
# delay notification for 30 minutes
notify(**kwargs, countdown=1800)
```

Custom provider:

```python
from notifications.utils import get_notification_model
from notifications.providers import BaseNotificationProvider
```

(continues on next page)

(continued from previous page)

```python
class DelayedNotificationProvider(BaseNotificationProvider):

    name = 'delayed_notifier'

    def send(self, payload):
        notification_id = self.payload['notification_id']

        notification = get_notification_model().objects.get(id=self.notification_id)
        if notification.read:
            return

        # send the notification
```

In this example, we abort the notification if the notification has been read when the provider is executed.

3.7.2 WebSockets

Unlike other django notification libraries that provide an API for accessing notifications, django-notifs supports websockets out of the box (thanks to *django-channels*). This makes it easy to send realtime notifications to your users in reaction to a new server side event.

If you're unfamiliar with django-channels. It's advised to go through the documentation so you can understand the basics.

3.7.3 Setting up the WebSocket server

This section assumes that you've already installed django-channels

Setup the consumer routing in your `asgi.py` file:

```python
import os

import django
from django.core.asgi import get_asgi_application
from channels.routing import ProtocolTypeRouter, URLRouter

from notifications import routing as notifications_routing


os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'yourapp.settings')

application = ProtocolTypeRouter({
    'http': get_asgi_application(),
    'websocket': URLRouter(notifications_routing.websocket_urlpatterns)
})
```

3.7.4 Notification channels

A simple WebSocket channel is provided:

`notifications.channels.DjangoWebSocketChannel`

Sample usage:

```
notif_args = {
    ...
    extra_data: {
        'context': {
            'channel_layer': 'default',
            'destination': 'group or channel_name',
            'message': {'text': 'Hello world'}
        }
    }
}
notify(**notif_args, channels=['websocket'])
```

3.7.5 Running the WebSocket server

`ASGI` is capable of handling regular HTTP and WebSocket traffic so you don't really need to run a dedicated WebSocket server but it's still an option.

see the channels deployment documentation for more information on the best way to deploy your application.

3.7.6 How to listen to notifications

You listen to notifications by connecting to the WebSocket URL.

The default URL is `http://localhost:8000/<settings.NOTIFICATIONS_WEBSOCKET_URL_PARAM>`

To connect to a WebSocket room (via JavaScript) for a user `john_doe` you'll need to connect to:

```
var websocket = new WebSocket('ws://localhost:8000/john_doe')
```

You can always change the default route by Importing the `notifications.consumers.DjangoNotifsWebsocketConsumer` consumer and declaring another route. If you decide to do that, make sure you use the `NOTIFICATIONS_WEBSOCKET_URL_PARAM` setting because the Consumer class relies on it

an example to prefix the URL with `/chat` would be:

```
from django.urls import path

from . import default_settings as settings
from .consumers import DjangoNotifsWebsocketConsumer

websocket_urlpatterns = [
    path(
        f'chat/<{settings.NOTIFICATIONS_WEBSOCKET_URL_PARAM}>',
        DjangoNotifsWebsocketConsumer.as_asgi()
    )
]
```

3.7.7 Authentication?

This is out of the scope of django-notifs for now. This might change in the future as django-channels becomes more mature. Hence, The WebSocket endpoint is unprotected and you'll probably want to roll out your own custom authentication backend if you don't make use of the standard Authentication backend.

3.7.8 Testing and Debugging

django-notifs comes with an inbuilt `'console'` delivery backend provider that just prints out the notification payload:

```
settings.NOTIFICATIONS_DELIVERY_BACKEND = 'notifications.backends.Console'
```

This can be helpful during development

PYTHON MODULE INDEX

INDEX