django-pushserver Documentation

Release 0.2.6

wlan slovenija

Sep 27, 2017

Contents

1 Contents **3**
1.1 Installation 3
1.2 Usage 4

2 Source Code and Issue Tracker **7**

3 Indices and tables **9**

django-pushserver is a push server for Django based on Leo Ponomarev's Basic HTTP Push Relay Protocol. Useful especially while locally developing Django applications using Nginx HTTP push module.

CHAPTER 1

Contents

Installation

Using pip simply by doing:

```
pip install django-pushserver
```

You should then add `pushserver` to `INSTALLED_APPS` and configure URLs used for HTTP push. For example:

```
PUSH_SERVER = {
    'port': 8001,
    'address': '127.0.0.1',
    'store': {
        'type': 'memory',
        'min_messages': 0,
        'max_messages': 100,
        'message_timeout': 10,
    },
    'locations': (
        {
            'type': 'subscriber',
            'url': r'/updates/([^/]+)',
            'polling': 'long',
            'create_on_get': True,
            'allow_origin': 'http://127.0.0.1:8000',
            'allow_credentials': True,
            'passthrough': 'http://127.0.0.1:8000/passthrough',
        },
        {
            'type': 'publisher',
            'url': r'/send-update/([^/]+)',
        },
    ),
}
```

Settings translate directly to settings of the py-hbpush package. Production settings should match those configured in Nginx.

You should add passthrough URLs to `urls.py`, matching URL configured in settings:

```
urlpatterns = patterns('',
    # ...

    url(r'^passthrough', include('pushserver.urls')),

    # ...
)
```

Passthrough URLs are not publicly accessible, so you should use `INTERNAL_IPS` to configure from which IPs they should be accessible. As you will probably run both Django development server and push server daemon on the same machine, this is probably simply:

```
INTERNAL_IPS = (
    '127.0.0.1',
)
```

When used in production where Nginx is making passthrough requests, it should match IP(s) on which you have Nginx running.

If you do not need or want passthrough just do not define it in `PUSH_SERVER` setting. Passthrough URLs and ‘ INTERNAL_IPS‘‘ setting are also not needed in this case.

Usage

Once installed, run the push server daemon (alongside Django development server):

```
./manage.py runpushserver
```

Or if you want that all requests are processed through push server (and non-push requests passed to Django), you can do:

```
./manage.py runpushserver --allrequests
```

(In this way you are not served static files auto-magically, like you might be used on Django development server, so you have to take care of that yourself. Auto-reloading on code change also doesn’t happen. Furthermore, by default it runs on port 8000 instead of 8001.)

Then you can push data to all clients subscribed to the given channel with simple HTTP request. If you for example want to push some JSON data (called update) you can use provided functions:

```
from pushserver.utils import updates

channel_id = 'some_channel_id'
data = {
    'type': 'answer',
    'value': 42,
}

updates.send_update(channel_id, data)
```

For JavaScript side you can use provided JavaScript code for processing pushed JSON data as it comes (it requires jQuery):

```
<script type="text/javascript" src="{{ STATIC_URL }}pushserver/updates.js"></script>
```

The code should be initialized with channels (and their URLs) to subscribe to. For that a Django template tag `channel_url` is provided:

```
{% load pushserver %}

<script type="text/javascript">
    $.updates.subscribe({
        'main_channel': '{% filter escapejs %}{% channel_url "some_channel_id" %}{%␣
↪endfilter %}'
    });
</script>
```

Provided code assumes that updates are JSON data, are a dictionary, and have a top-level value named `type` by which you can register different update processors in your JavaScript code. To continue the example:

```
function updateAnswer(data) {
    // data.value == 42
}

$.updates.registerProcessor('main_channel', 'answer', updateAnswer);
```

Arguments to `$.updates.registerProcessor` are the name of the channel as you have given to `$.updates.subscribe`, the `type`, and a processor function which will be called with given data everytime data arrives.

If you want to process passthrough requests clients are making when subscribing or unsubscribing, you can connect to provided signals:

```
from django import dispatch

from pushserver import signals

@dispatch.receiver(signals.channel_subscribe)
def process_channel_subscribe(sender, request, channel_id, **kwargs):
    print "Subscribed", request.user, channel_id

@dispatch.receiver(signals.channel_unsubscribe)
def process_channel_unsubscribe(sender, request, channel_id, **kwargs):
    print "Unsubscribed", request.user, channel_id
```

Because user credentials were being passed through in this example, Django session and authentication middlewares should work as expected, populating `request.user`.

Be aware that for each sent update, clients unsubscribe and soon afterwards subscribe again so many signals could be triggered in a rapid succession. Because of this signal receivers should be very light-weight.

CHAPTER 2

Source Code and Issue Tracker

For development GitHub is used, so source code and issue tracker is found there.

CHAPTER 3

Indices and tables

- genindex
- search