A minimal Websockets setup with Django in production

2020-08-02

That's me wiring up my cool JS grid to autoload data changes via server side push. Yeah! Image credit: thedisgruntledsherpaproject.com

Use Case

I have a Handsontable implementation for an underlying database table. I.e. a "JavaScript data grid that looks and feels like a spreadsheet".

A requirement came up. Obvious in hindsight.

Changes made to cells in one sheet should be reflected in the same sheet for other users. When the sheet is open in other browser tabs/windows.

This calls for "server side push" using web sockets. I.e. the server needs to push a notification to open "clients".

Another way to do this would be to have client browsers Ajax-polling for changes. But that would be wasteful. Let's only update the sheet when valid changes are saved!

What does *minimal* mean in this case?

This application is used by a team internally. Usage not exceeding ten concurrent users. The implications of this:

- *One* process to serve web socket requests is enough.
- No real performance testing was done.
- No consideration of alternatives to `daphne` such as uvicorn or starlette. I picked up `daphne` because it came up "first on the list of alternatives". That's it!
- No need to handle websocket interactions asynchronously in my case. You can read more about going sync vs async with websockets in Django channels here.

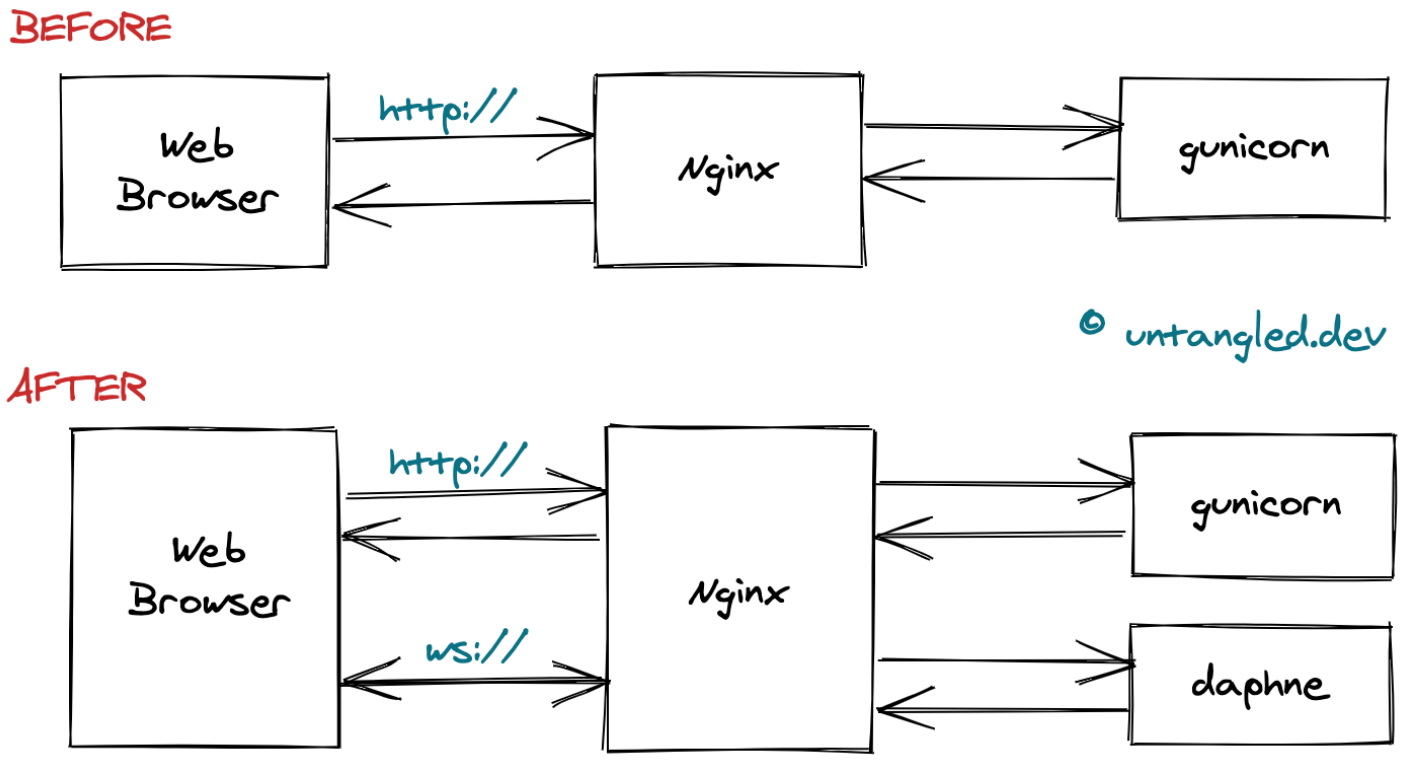
This configuration remains unchanged until problems crop up. Because "premature optimisation is the root of all evil".

Blueprint: Before & After

Note: `http` request protocol below refers both to `http` and `https`. Same applies to `ws` and `wss`. Assume that for local development the protocol is unsecure, whilst secure in production.

Before introducing websockets, the web browser made an `http` request to Nginx. At this point Nginx serves the request using `gunicorn`, hitting Django[1].

After adding websockets in the mix, Nginx still serves `http` requests. But it's now able to serve `ws` requests by talking to `daphne`. In this case you can replace `daphne` with any other Websocket termination server:



The "new" item in the building blocks above is therefore daphne:

> Daphne is a HTTP, HTTP2 and WebSocket protocol server for ASGI and ASGI-HTTP, developed to power Django Channels.
>
> It supports automatic negotiation of protocols; there's no need for URL prefixing to determine WebSocket endpoints versus HTTP endpoints.

The `daphne` component can be replaced with alternatives as uvicorn or starlette.

The other item of note is that `ws://` connections are an "open" connection. "Data" travels in both directions *along* the same socket. As opposed to `http://`.

Code changes

`daphne` is part of the Django Channels effort:

> Channels augments Django to bring WebSocket, long-poll HTTP, task offloading and other async support to your code, using familiar Django design patterns and a flexible underlying framework that lets you not only customize behaviours but also write support for your own protocols and needs.

I've followed the docs. The installation and excellent tutorial sections helped me get everything to work locally.

For completeness' sake, these code changes are for an installation using these package versions:

```
channels==2.4.0
channels-redis==3.0.1
Django==3.0.8
redis==3.5.3
```

in a Python `3.7.5` virtualenv.

The changes needed:

1. `settings.py` changes. These:
    - add `channels` to `INSTALLED_APPS`, and
    - configure `channels` to route websocket requests to the main channels entrypoint
2. Routing code changes:
    - main project-level routing entrypoint
    - app level entrypoint(s), in this example using just one example `myapp` as app
3. The consumer that hosts all the event-handling and message sending logic our app needs to implement.

1. settings module changes

Added `channels` as *the first* app in my project's list of `INSTALLED_APPS`. Why first?

> Please be wary of any other third-party apps that require an overloaded or replacement `runserver` command. Channels provides a separate `runserver`

command and may conflict with it. An example of such a conflict is with `whitenoise.runserver_nostatic` from whitenoise. In order to solve such issues, try moving channels to the top of your `INSTALLED_APPS` or remove the offending app altogether.

Then added this new setting for `channels` app to use:

```
# CHANNELS
ASGI_APPLICATION = 'proj.routing.application'
CHANNEL_LAYERS = {
    'default': {
        'BACKEND': 'channels_redis.core.RedisChannelLayer',
        'CONFIG': {
            'hosts': [('127.0.0.1', 6379)],
        },
    },
}
```

Note that I already had `redis` installed for caching and the application's existing task queue with Huey.

2. Routing changes

I usually call the "default" app `proj`. This makes it obvious that the app is a container for project-wide items. As is the case with this new `routing` module. It contains the ProtocolTypeRouter that serves as main entry point for the ASGI application. `proj/routing.py` contents:

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

application = ProtocolTypeRouter({
    # (http->django views is added by default)
    'websocket': AuthMiddlewareStack(
        URLRouter(
            myapp.routing.websocket_urlpatterns
        )
    ),
})
```

`myapp` is the test app used for this exmaple. The top-level router above contains a reference to a `myapp.routing` module. This URLRouter routes `http` or `websocket` type connections via their HTTP path. `myapp/routing.py` contains the below:

```
from django.urls import re_path

from myapp import consumers

websocket_urlpatterns = [
    re_path(r'ws/sheet/(?P<sheet_name>\w+)/$', consumers.SheetConsumer),
]
```

Note how `channels` allows us to structure our web socket URLs in the already familiar format we're used for standard `urls.py` configuration[2].

3. The consumer

The final module that needs adding is the consumer. In `channels`, consumers:

- Structure your code as a series of functions to be called whenever an event happens, rather than making you write an event loop.
- Allow you to write synchronous or async code and deals with handoffs and threading for you.

`myapp/consumers.py` implements a `SheetConsumer` class which extends WebsocketConsumer:

```
import json
from asgiref.sync import async_to_sync
from channels.generic.websocket import WebsocketConsumer


class SheetConsumer(WebsocketConsumer):

    def connect(self):
        self.sheet_name = self.scope['url_route']['kwargs']['sheet_name']
        self.sheet_group_name = 'sheet_%s' % self.sheet_name

        # Join sheet group
        async_to_sync(self.channel_layer.group_add)(
```

```
            self.sheet_group_name,
            self.channel_name
        )
        self.accept()

    def disconnect(self, close_code):
        # Leave sheet group
        async_to_sync(self.channel_layer.group_discard)(
            self.sheet_group_name,
            self.channel_name
        )

    # Receive message from WebSocket
    def receive(self, text_data):
        text_data_json = json.loads(text_data)

        # Send sheet_name to sheet group
        async_to_sync(self.channel_layer.group_send)(
            self.sheet_group_name,
            {
                'type': 'refresh_sheet',
                'sheet_name': text_data_json['sheet_name'],
                'object_id': text_data_json['object_id'],
                'column_index': text_data_json['column_index'],
                'new_value': text_data_json['new_value'],
                'broadcaster_id': text_data_json['broadcaster_id'],
            }
        )

    # Receive message from sheet group
    def refresh_sheet(self, event):
        # Send sheet_name to WebSocket
        self.send(text_data=json.dumps({
            'sheet_name': event['sheet_name'],
            'object_id': event['object_id'],
            'column_index': event['column_index'],
            'new_value': event['new_value'],
            'broadcaster_id': event['broadcaster_id'],
        }))
```

The above is based on the Write your first consumer tutorial section. Instead of chat messages, the data is about a sheet's cell updates. Updates that need to be applied *for the same sheet* open in other browser tabs/windows.

I'm passing the user ID of the authenticated user in `broadcaster_id`. To be able to tell which user "triggered" the websocket message being "broadcasted".

Give it a try locally

This is another great feature of `channels`. Not even your usual `manage.py runserver` workflow needs to change. Just note the new item in your default `runserver` output when it starts:

```
$ ./manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
August 01, 2020 - 16:07:41
Django version 3.0.8, using settings 'proj.settings.local'
Starting ASGI/Channels version 2.4.0 development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

The socket handshakes are also shown in the output:

```
WebSocket HANDSHAKING /ws/sheet/sheet1/ [127.0.0.1:65181]
WebSocket CONNECT /ws/sheet/sheet1/ [127.0.0.1:65181]
```

Awesome! Let's deploy!

Deployment notes

Not so fast 😊

I followed the `channels` documentation here alongside Django's own documentation on deploying ASGI applications here. But I applied ~~two~~ three tweaks that I'd rather explain.

daphne command tweak

I experienced the `CRITICAL Listen failure: [Errno 88] Socket operation on non-socket` exception described here.

This was fixed by following the suggestion to remove the `-fd 0` switch. This switch is suggested by default in the channels docs.

In the current use case I do not need to use this switch. Because I do not need to bind multiple Daphne instances to the same port my production instance. In case I do, I will need to change my structure (see next section) to have `daphne` called directly from supervisor. Rather than via bash script.

asgi.py tweak

I had implemented `proj/asgi.py` as described in the channels docs here. This works fine locally. But it led to this exception described here when executing web socket requests in production. I changed the `proj/asgi.py` as described in this stackoverflow answer which makes it have this content:

```
import os
import django
from channels.routing import get_default_application

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

django.setup()

application = get_default_application()
```

This change replaces usage of `django.core.asgi.get_asgi_application` with `channels.routing.get_default_application`. The stackoverflow answer above is supported as per `channels`' docs here.

I do not know whether this fixes things *properly*. Should I should have done something else? It appears to be a small incompatibility between `channels` and Django docs. `channels` docs suggest creating `asgi.py` from scratch. While Django 3.0.8 auto-created `asgi.py`.

If you have a better resolution to this please let me know (comment below).

redis version

Recall that my configuration is using channels_redis as backing store.

Since my production application runs on Ubuntu 18.04 LTS, default `apt-get` redis version was `4.0.1`.

This resulted in this weird `BZPOPMIN - "ERR unknown command 'BZPOPMIN'"` error. This is because redis version 5 or higher is needed.

Therefore please upgrade redis for your configuration. In my case I've followed the quickstart docs, especially the "Installing Redis more properly" section.

On to the production config files!

Deployment - resulting configuration

My configuration's components:

- An executable bash script runs `daphne`. I use this to be able to run and test `daphne` directly in the Django project's virtualenv.
- A supervisor `conf` file to have this bash script process managed by supervisor.
- Nginx, of course.

Bash script

`start_daphne.bash` contents. Remember to `chmod +x` your bash script.

```
#!/bin/bash

NAME="myproject-daphne"  # Name of the application
DJANGODIR=/home/ubuntu/webapp/myproject/proj  # Django project directory
DJANGOENVDIR=/home/ubuntu/webapp/myprojectenv  # Django project env

echo "Starting $NAME as `whoami`"

# Activate the virtual environment
cd $DJANGODIR
source /home/ubuntu/webapp/myprojectenv/bin/activate
source /home/ubuntu/webapp/myproject/proj/.env
export PYTHONPATH=$DJANGODIR:$PYTHONPATH

# Start daphne
exec ${DJANGOENVDIR}/bin/daphne -u /home/ubuntu/webapp/myprojectenv/run/daphne.s
```

Supervisor

File located at: `/etc/supervisor/conf.d/daphne.conf`. Remember to create the log file directories.

```
; ================================
;  daphne supervisor
; ================================

[program:daphne]
command = /home/ubuntu/webapp/start_daphne.bash  ; Command to start app

user = ubuntu   ; User to run as
numprocs=1

autostart=true
autorestart=true

redirect_stderr=true
stdout_logfile = /home/ubuntu/webapp/logs/daphne/access.log  ; Where to write ac
stderr_logfile = /home/ubuntu/webapp/logs/daphne/error.log  ; Where to write err
stdout_logfile_maxbytes=50MB
stderr_logfile_maxbytes=50MB
stdout_logfile_backups=10
stderr_logfile_backups=10
environment=LANG=en_US.UTF-8,LC_ALL=en_US.UTF-8  ; Set UTF-8 as default encoding
```

Nginx

Nginx configuration references I used: channels deployment docs and this answer on stackoverflow. Follow those links to understand what I did.

Relevant Nginx config contents:

```
upstream ws_server {
  server unix:/home/ubuntu/webapp/myprojectenv/run/daphne.sock fail_timeout=0;
}

upstream gunicorn_server {
  server unix:/home/ubuntu/webapp/myprojectenv/run/gunicorn.sock fail_timeout=0;
}

...

```

```
server {
  ...

  location /ws/ {
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection "upgrade";
    proxy_redirect off;
    proxy_pass http://ws_server;
  }

  location / {
    ...

    if (!-f $request_filename) {
        proxy_pass http://gunicorn_server;
        break;
    }
  }
}
```

Note the newly-added `ws_server`-related parts.

A note Markup/Javascript code

This is not configuration as such. But as you can see the whole tutorial did not tackle `ws` and `wss` usage. One reason is that in this project's case the SSL certificate part is not handled by Nginx. Since the project is using Cloudflare SSL, Cloudflare takes care of it even "before Nginx".

The only `ws` vs `wss` logic I have is done at client-side level. This allows the same code to use the correct protocol locally and in production. The code sets up the connection depending on the current `http` protocol in use:

```
    ...

    if (window.location.protocol == 'https:') {
      wsProtocol = 'wss://'
    } else {wsProtocol = 'ws://'}

    sheetSocket = new WebSocket(
      wsProtocol + window.location.host
```

```
        + '/ws/sheet/' + sheetName + '/'
    );

    ...
```

Conclusion

Please let me know (in the comments below) whether anything I've done is wrong or I can improve it.

This was my first experience with Websockets and Django together. And it was pleasant one. The few "conflicting docs" issues described above, although blocking, were kinda expected.

Credits: Diagram above drawn using excalidraw.com.

1. In practice, in some cases, Nginx is able to serve the request itself. Example, for static files or resources cached an Nginx level. Avoiding going into this to keep things simple. ↩
2. At the time of writing `re_path()` is used due to limitations in URLRouter. ↩

Tags: Django, Nginx

Subscribe via Atom feed, Twitter, or email:

Email Address

Subscribe

I won't spam you. I'm too lazy. And I hate spam. Maximum one email per month. Thanks!