The Fool

- Blog
- projects

real-time progress updates with django channels
Sep 12, 2017
13 minute read

- channels
- websocket
- django

Abstract

What follows is a simple demo for using the Delay Server in Django Channels to deliver asynchronous real-time progress updates over WebSockets. Our demo backend will execute a long-running 'background task', and then send progress updates at regular intervals to a WebSocket client. We'll also use channels to distribute our server load over multiple queues and worker processes.

Full project code here

Why Django Channels?

Not wat, but why. For pretty much the whole history of software, there was a handy solution to most problems with *performance*: just wait a year for CPU clock rates to improve. Bingo! Your program suddenly performs twice as fast! But now the Age of Single-Threadedness is in decline, as CPU speed increases have begun mellowing out. To improve our performance we

need to make actual changes to the code and *architecture* of our software. We get our speed-up by running multiple threads of execution in parallel. Enter Channels.

Django Channels takes a familiar tack: a dedicated pool of worker processes consuming a task queue. This is a *message passing* paradigm. Data is serialized and stuffed in a mailbox. This is *not* shared memory concurrency. The benefit of this worker-process-pool & message-queue technique is that it is dead simple to implement (relative to 'shared mutable memory' models, at least). We create a task, push it onto a queue, and eventually a free process will scoop it up and do the work.

Django Channels is a structural change in Django's web-server innards. It breaks up the Django process into two teams: we have a dedicated server that receives requests and subsquently *enqueues* tasks, and a server (or pool thereof) that *consumes* those tasks. Additionally, since a 'task' and a 'consumer' are pretty generic notions, Django is no longer just built for the traditional HTTP request/response pipeline.

This is nothing novel. In fact, most Django production environments are buoyed with software that follows this pattern: nginx for managing a pool of HTTP consumers, Gunicorn for managing a pool of Python processes, and Celery for managing a different pool of Python processes. Maybe in the future of the Channels project we won't have a use for so many auxillary worker-pool managers?

Oh, and WebSockets

I forgot to mention WebSockets – Django Channels also provides a pretty neat way to do WebSocket-related things. That's what we're about to get into.

The Problem: Real-Time status updates at regular intervals

I was recently charged to write a web-server that handles requests from its web-client for starting a long-running task on *another* server, and then sending the results back once it's complete. The web-server is something of a proxy, brokering events between a backend service and a web-client. It listens to a client, and sends the request for a task on through to the service. But this backend service *also* provides a way to retrieve the progress of a task it's working on. Since I can get the progress of a task, I also want to update the client at *regular intervals* with the status.

One way to get regular updates is to let the *client* drive by making many HTTP requests over time, polling our web-server. The downside of this is processing overhead for the backend (not only with respect to the bulky HTTP packets, but also that every request needs to be authenticated afresh). Rejected!

A better way is to use WebSockets. That way, the client and the web-server can create a session, and the server can push updates straight to the client. No need for the client to request them. Nice, light, declarative ⋯ and fun!

What we need now is just a way to get our Django app to poll the backend service periodically, and then send the client an update. And, no, we are not going to go this route in our Django code:

```
# this is wrong, wrong, wrong
while not complete:
  sleep(1)
  current_status = poll_service(task_id)
  update_client(current_status)
  complete = current_status['complete']
```

What we want to do is *queue* a slightly delayed task that polls the backend service. We won't be putting threads to sleep. Message queuing is also nice, light, declarative ⋯ and fun!

Implementation Decision

To play with WebSockets in Django, there are options. But it seems pretty safe to assert that Channels is going to be the *de jure* solution, as well as *de facto*.

But what about our timeout intervals?

Typically, I'd reach for Celery to address a problem like this. With Celery you can specify a 'countdown' for a task, so that it won't be consumed until a set time into the future.

Celery would be a perfectly fine solution! But it turns out that we can accomplish the feature with Django Channels alone. In this case, we do not need to bring in the whole Celery project. *Nota bene: Channels is **not** (yet?) a replacement for Celery, but they do overlap in places.*

Whereas Celery has its 'countdown' attribute for its async tasks, Channels has a delay server for handling 'delayed' tasks (i.e. tasks whose execution is meant to occur after at least a certain amount of time into the future).

So, no Celery needed! We can get *both* WebSocket goodness *and* regular timeout intervals with the same Django library. What a world!

Let's build it

We're going to build a basic client that opens a WebSocket connection with our Django app, requests that a long-running task begin, and then gets hit with periodic updates until the task completes.

Alert: this example is going to use Docker and docker-compose to manage the multiple servers we need. If you don't want to use Docker, there will only need to be slight modifications to the configuration of the Django app.

Since we're rocking real-time progress, let's name our app ProgRock. Go ahead and clone the repo:

```
git clone https://github.com/the-fool/prog-rock
```

Let's first look at the requirements.txt file. There are some interesting Channels-specific thing in there. They resulted from:

```
pip install channels
pip install asgi_redis
```

The Channels package brings with it the Daphne server. And we also needed to install asgi_redis because we are going to use Redis as our message-queue engine.

Now take a look at the docker-compose.yml:

```
version: '2'

services:
 redis:
  image: redis:latest

 django:
  build:
   context: .
   dockerfile: ./docker/django/Dockerfile
  depends_on:
   - redis
  volumes:
   - .:/app
  ports:
   - "8000:8000"
  command: python manage.py runserver 0.0.0.0:8000

 django-delay:
  build:
   context: .
   dockerfile: ./docker/django/Dockerfile
  depends_on:
   - redis
  volumes:
   - .:/app
  command: python manage.py rundelay
```

We have 3 separate services, but 2 of them are nearly identical. The first Django service, which executes the 'runserver' command, is going to be the 'main' public facing web-server. The

second Django service is our 'delay server', whose mission is to handle the timeouts and enqueing of delayed tasks.

If you don't recognize the `rundelay` argument passed to `manage.py`, don't fret! It's a new feature added when we hook in the Django Channels app. In our `settings.py` file you'll find:

```
INSTALLED_APPS = [
    . . .

    # The Channels project apps
    'channels',
    'channels.delay',

    . . .
]
```

Simply registering the Channels app is all it takes to transform classical Django into the new distributed worker architecture!

But Channels won't do much without some configuration. We need to add Channels-specific plumbing. Also in `settings.py`:

```
CHANNEL_LAYERS = {
    "default": {
        "BACKEND": "asgi_redis.RedisChannelLayer",
        "CONFIG": {
            "hosts": [("redis", 6379)],
        },
        "ROUTING": "config.routing.channel_routing",
    },
}
```

This is pretty simple. We need to identify three things to get off the ground: our backend implementation for the message queue, configuration for that backend, and configuration for the rules of our channels (routing). In our case, we're using Redis as the backend. As far as I can surmise, Redis is currently the most popular pick, and if there is any downside to using Redis instead of another message-queue broker, I haven't yet discovered it. Our 'hostname' for Redis is simply the name of the Dockerized Redis service, which in our yaml config is `redis`.

Routing is very similar to the url rules we would specify for our HTTP Django apps. The way I see it, channels routing is just a slightly more generalized abstraction of the same idea behind a Django url rule. A Channels route is not much more than a rule for turning raw events into a message, placing that message into a queue, and a rule for what sub-routine consumes which queue.

With url rules in Django we typically have a *root level* table which imports in *app specific* sub routes. We follow the same design principle with Channels routing. Our root `routing.py` looks like this:

```
from channels.routing import include

channel_routing = [
  include('progrock.routing.public_routing', path=r'^/prog/'),
  include('progrock.routing.internal_routing')
]
```

And then our app-specific sub-module progrock/routing.py in the ProgRock app is is this:

```
from channels.routing import route
from .consumers import ws_receive, ws_connect, worker

public_routing = [
  route('websocket.connect', ws_connect),
  route('websocket.receive', ws_receive),
]

internal_routing = [
  route('prog-rocker', worker)
]
```

We define an *internal* and a *public* routing as way to distinguish the sort of routes that relate to requests made from an external client, and those other routing rules which are only inside our app itself. Notice that the public routes are prefixed by a path regex – this is a way to namespace WebSocket events to a particular app. *(In this demo there is only 1 WebSocket receiver – so this namespacing is not very useful, but it's good practice anyway!)*

Another thing to note is that route definitions are just a 2-part tuple – a string key and a sub-routine. The 'channel' is named by the string key. Django Channels provides out-of-the-box a few conventional names for common channels. Two such conventional names are the websocket.connect and websocket.receive channels. More on this in a minute.

Our third route defines an all-custom channel: the prog-rocker channel. All we need to do is name it, and it comes into existence! This channel is going to be for a special worker responsible for doing low-priority things such as sending progress updates and polling the backend service. We separate our queues for *receiving* messages from the queue for *responding* to messages. This is because we don't want slow responses to get in the way of our receiver-workers.

To see how these channels gear into the web-client, let's look at the frontend code. First, the relevant HTML:

```
<button id="go-button">Rock Out</button>
<div>
 Progress <span id="prog-val">0%</span>
</div>
```

Just a button and some text indicating the progress. What we are going to do is fire up a task when the user clicks the button, and then update the progress in the markup. Here is all of the

JavaScript:

```
var ws_scheme = window.location.protocol == "https:" ? "wss" : "ws";
var ws_path = ws_scheme + '://' + window.location.host + '/prog/';

var socket = new WebSocket(ws_path);

socket.onmessage = onmessage;
document.getElementById('go-button').onclick = start;

function onmessage(msg) {
 var value = document.getElementById('prog-val');
 var data = JSON.parse(msg.data);

 if (data.progress) {
  value.innerHTML = data.progress + '%';
 }

 if (data.complete) {
  value.classList.add('complete');
 }
}

function start() {
 var msg = {
  action: 'start'
 };

 socket.send(JSON.stringify(msg));
}
```

The path of the WebSocket matches with the namespacing rule in our root channel routing. WebSocket requests with the path /prog/ go to our ProgRock app. After establishing a WebSocket connection, the client reacts to future messages with appropriate updates. Relevant messages we listen to are going to be of type {progress: number, complete: boolean}. Not the most robust way to serialize state – but it's good enough for a demo!

Now to the main course: the consumers.py module in our ProgRock app. First, the two simple consumer routines for connecting and receiving WebSocket events:

```
import json
from channels import Channel

def ws_connect(msg):
  msg.reply_channel.send({'text': json.dumps({"accept": True})})


def ws_receive(message):
  data = json.loads(message['text'])
  reply_channel = message.reply_channel.name

  if data['action'] == 'start':
```

```
    Channel('prog-rocker').send({
      'reply_channel': reply_channel,
      'command': 'start',
    })
```

The ws_connect consumer reacts to new WebSocket connections. For our case, it actually does not need to do anything. We immediately send back an acknowledgement message for debugging purposes. All other 'sends' back to the web client are going to queued up for a dedicated consumer (below).

The ws_receive is for messages sent from the web-client over a WebSocket to our app. Looking up at the javascript, there is only one such message: {action: 'start'}. In a more complicated app, our Django code would likely have a big 'switch' of if -- elif -- elif parsing the message action. In our single switch, the consumer catches the 'start' action and reacts.

Starting a new long-running task is very low-priority, so we want to bail from this consumer sub-routine without going through the trouble of firing up the task. That is why we *pass this message onto a worker queue*. The Channel() constructor is used to call up a registered channel by name, after which you can call methods on it. We are going to .send() a message to that channel. Our internal prog-rocker API is looking for a `command to specify what it should do, and it also needs a reply_channel so that it knows how to communicate with the open WebSocket connection to the client.

So far so good. We handle WebSocket messages with something like a proxy. It switches on an action (e.g. 'start'), and does no more work than sending a message to another queue. When sychronicity does not truly matter, it is generally good practice to opt for the asynchronous path. We don't *need* to respond right away, so we queue up the response and move on.

Now comes the complicated part. Our special worker sub-routine. Without further ado, here is the sub-routine from the consumers.py module *in toto*:

```
def worker(msg):
  def delayed_message(reply_channel, progress, action):
    return {
      'channel': 'prog-rocker',
      'content': {
        'reply_channel': reply_channel,
        'action': action,
        'progress': progress,
      },
      'delay': 200
    }

  def delayed_send(msg):
    Channel('asgi.delay').send(msg, immediately=True)

  def send_to_ws(reply_channel, content):
    Channel(reply_channel).send({'text': json.dumps(content)})
```

```
    def send_progress(reply_channel, progress):
        send_to_ws(reply_channel, {'progress': progress})

    action = msg.content['action']
    reply = msg.content['reply_channel']

    if action == 'start':
        # start a long running background task!
        # (this is just a mockup, using number to represent the progress of a background task)
        progress = 0

        send_progress(reply, progress)

        new_msg = delayed_message(reply, progress, 'continue')
        delayed_send(new_msg)

    elif action == 'continue':
        # check in on the background task
        # (we are simulating a task by incrementing a number)
        progress = msg.content['progress'] + 3
        if progress >= 100:
            send_to_ws(reply, {'complete': True, 'progress': 100})
        else:
            send_progress(reply, progress)
            new_msg = delayed_message(reply, progress, 'continue')
            delayed_send(new_msg)
```

There's a lot going on here. But most of the code above is a collection of inner helper funcitons. The procedural stuff is near the bottom. Again, we switch on an action. When the action is 'start', we want to execute the long-running background task on the third party server. For simplicity in this mockup, we'll just assign a variable to 0.

Now that we have a brand new task started, we send a message through the WebSocket back to the client. This in itself is a great accomplishment! We've made a beautiful architecture where we receive messages and queue up their responses, keeping things lean and mean.

An interesting part is how we implement the interval delay for polling. Remember that we registered the app channels.delay in the INSTALLED_APPS list? This app provides a channel 'asgi.delay'. Messages to this channel are meant to be delayed in execution. We wrap the 'real' message in a structure that includes the time to delay, and also the name of the channel to send the message when it's time has come. The message sent to the delay server is:

```
{
  'channel': 'prog-rocker',
  'content': {
    'reply_channel': reply_channel,
    'action': action,
    'progress': progress,
  },
  'delay': 200
}
```

The `content` is the ‘real’ message. The `delay` is the time in milliseconds. And the `channel` is the channel. Notice that we are recursing! This is a very common pattern in message-passing architectures. Also note that our consumer sub-routine only has *stack-local* state, so we need to pass in the `reply_channel` again. We can’t, for example, save it in a global variable.

After we recurse the first time, the action is to `'continue'`. In our worker, the switch polls the backend service (again, this ‘polling’ is just mocked up), and then it checks whether or not the task has completed, and sends the appropriate message down the line.

Lather, rinse, recurse!

Conclusion

We’ve used Django Channels to handle WebSocket communication. And we’ve used Django Channels to implement delayed tasks; we queued up a task to run after a specified point in the future. We also put together a very clean and scalable separation of our receive & response workers. By breaking those features up into small consumable tasks, we increase the parallelism of our app. Welcome to real-time asynchronous Django!

user9001 • 2 years ago

thanks for these great example.
do you have a example to implement a long running background task? is it possible to stop the long running background task when I send a web socket message from client to the worker? What happens when more user opens the website? does each user have his own background task?