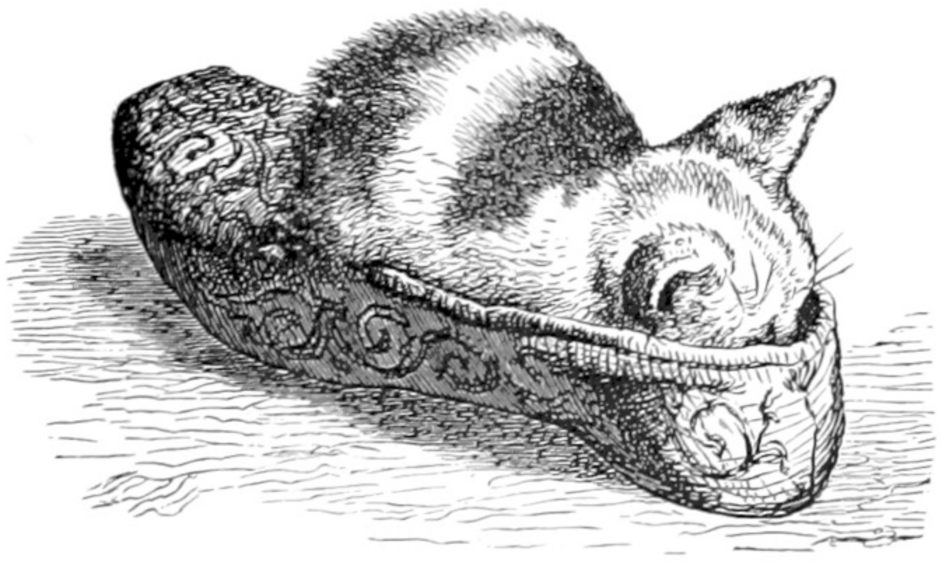
A Django REST API in a Single File

2020-10-15



I previously covered writing a Django application in a single file, for both synchronous and asynchronous use cases. This post covers the angle of creating a REST API using Django in a single file.

Undeniably, REST API's are a very common use case for Django these days. Nearly 80% of this year's Django community survey respondents said they use Django REST Framework (DRF). DRF is great for building API's and provides many of the tools you'd want in a production-ready application. But for building a very small API, we can get by solely with tools built into Django itself.

Without further ado, our example application is below. You can save it as `app.py`, and run it with `python app.py runserver` (tested with Django 3.1). An explanation follows after the code:

```
import os
import sys
from dataclasses import dataclass

from django.conf import settings
from django.core.wsgi import get_wsgi_application
from django.http import HttpResponseRedirect, JsonResponse
from django.urls import path
from django.utils.crypto import get_random_string

settings.configure(
    DEBUG=(os.environ.get("DEBUG", "") == "1"),
    ALLOWED_HOSTS=["*"],  # Disable host header validation
    ROOT_URLCONF=__name__,  # Make this module the urlconf
    SECRET_KEY=get_random_string(
        50
    ),  # We aren't using any security features but Django requires this set
ting
    MIDDLEWARE=["django.middleware.common.CommonMiddleware"],
)


@dataclass
class Character:
    name: str
    age: int

    def as_dict(self, id_):
        return {
            "id": id_,
            "name": self.name,
            "age": self.age,
        }


characters = {
    1: Character("Rick Sanchez", 70),
    2: Character("Morty Smith", 14),
}


def index(request):
    return HttpResponseRedirect("/characters/")


def characters_list(request):
    return JsonResponse(
        {"data": [character.as_dict(id_) for id_, character in characters.it
ems()]}
    )


def characters_detail(request, character_id):
    try:
        character = characters[character_id]
    except KeyError:
        return JsonResponse(
            status=404,
            data={"error": f"Character with id {character_id!r} does not exi
st."},
        )
    return JsonResponse({"data": character.as_dict(character_id)})


urlpatterns = [
    path("", index),
    path("characters/", characters_list),
    path("characters/<int:character_id>/", characters_detail),
]

app = get_wsgi_application()

if __name__ == "__main__":
    from django.core.management import execute_from_command_line

    execute_from_command_line(sys.argv)
```

Neat, just 73 lines, or 63 not counting imports.

The first thing we do, following imports, is to call `settings.configure()` with the minimum configuration to get Django running. I covered most of these settings in my first single-file app post which I won't repeat too much here.

One extra thing we're using compared to that post is `CommonMiddleware`, one of Django's many "included batteries". In its default configuration it will redirect URL's not ending with a slash ("/") to those with one, useful for getting users to their intended content.

Second, we define some static data for our API to serve, using dataclasses (new in Python 3.7). These are great for storing and serving a small amount of unchanging data. At some point we'd want to move to using a database, but for our purposes it is easier to avoid setting this up.

(I've also shown my bad taste by making this a Rick and Morty character API.)

Third, we define three views:

- `index` redirects to the character list URL, as that's our only data type in the API. If we expanded our API, we might want to show a "front page".

- `characters_list` returns a list of characters. If our list of characters grew large, we might want to paginate this to return only slices of characters at a time.
- `characters_detail` returns the representation of a single character. This also has an error path for when we're given an ID that doesn't match.

Fourth, we map URL's to our views in the `urlpatterns` list.

Fifth, we create the WSGI application object, which allows us to deploy the application. For example, if we'd saved this file as `app.py`, we could run it on a production server with `gunicorn app:app`.

Sixth, we introduce `manage.py` style handling when the module is run as `"__main__"`. This allows us to run the application with `python app.py runserver` locally.

Trying It Out

Here's a sample of using that API with httpie, a neat command-line tool for making HTTP requests.

First, hitting the index URL:

```
$ http localhost:8000
HTTP/1.1 302 Found
Content-Length: 0
Content-Type: text/html; charset=utf-8
Date: Thu, 15 Oct 2020 21:05:09 GMT
Location: /characters/
Server: WSGIServer/0.2 CPython/3.8.5
```

This redirects us to `/characters/` as expected. Fetching that, we see the JSON dat for both characters:

```
$ http localhost:8000/characters/
HTTP/1.1 200 OK
Content-Length: 101
Content-Type: application/json
Date: Thu, 15 Oct 2020 21:05:15 GMT
Server: WSGIServer/0.2 CPython/3.8.5

{
    "data": [
        {
            "age": 70,
            "id": 1,
            "name": "Rick Sanchez"
        },
        {
            "age": 14,
            "id": 2,
            "name": "Morty Smith"
        }
    ]
}
```

We might try fetching Morty's page:

```
$ http localhost:8000/characters/2
HTTP/1.1 301 Moved Permanently
Content-Length: 0
Content-Type: text/html; charset=utf-8
Date: Thu, 15 Oct 2020 21:05:19 GMT
Location: /characters/2/
Server: WSGIServer/0.2 CPython/3.8.5
```

Aha! We didn't add the trailing slash, so our request has been redirected. Fetching the complete URL, we see Morty's data:

```
$ http localhost:8000/characters/2/
HTTP/1.1 200 OK
Content-Length: 53
Content-Type: application/json
Date: Thu, 15 Oct 2020 21:05:21 GMT
Server: WSGIServer/0.2 CPython/3.8.5

{
    "data": {
        "age": 14,
        "id": 2,
        "name": "Morty Smith"
    }
}
```

Success!

Tests

These days I can't write a blog post without mentioning testing. We can use Django's built-in test framework to write some quick tests that cover all our endpoints.

Save this code in a file called `tests.py`, in the same folder as the application:

```
from django.test import SimpleTestCase


class AppTests(SimpleTestCase):
    def test_index(self):
        response = self.client.get('/')

        self.assertEqual(response.status_code, 302)
        self.assertEqual(response['Location'], '/characters/')

    def test_list(self):
        response = self.client.get('/characters/')

        self.assertEqual(response.status_code, 200)
        self.assertEqual(
            response.json()['data'][0],
            {"id": 1, "name": "Rick Sanchez", "age": 70},
        )

    def test_detail(self):
        response = self.client.get('/characters/1/')

        self.assertEqual(response.status_code, 200)
        self.assertEqual(
            response.json()['data'],
            {"id": 1, "name": "Rick Sanchez", "age": 70},
        )
```

These tests use Django's test client to make requests and then assert on the response status code and content. Because we don't have a database, the tests can inherit from the faster `SimpleTestCase`, rather than using `TestCase` which adds some database management. We can run these tests using `app.py` like so:

```
$ python app.py test
System check identified no issues (0 silenced).
...
----------------------------------------------------------------------
Ran 3 tests in 0.004s

OK
```

Great!

Fin

I hope this has helped you figure out a way into building small API’s with Django. If your API takes off, do check out Django REST Framework!

For a longer tutorial that uses the database, check out Curtis Maloney’s post.

—Adam

Related posts:

- Django versus Flask with Single File Applications
- A Single File Asynchronous Django Application

Tags: django