Django Sync or Async?

Python has added the asyncio library since its version 3.4 release. Django started supporting the asynchronous ("async") view on version 3.0. In case you are wondering whether it's worth refactoring the synchronous view to the async view, I hope this article will provide the answer you are looking for. In this article, we are going to dive into the actual implementations and performance tests. Please note that this article assumes that you already knew the basic concept of Django and Docker.

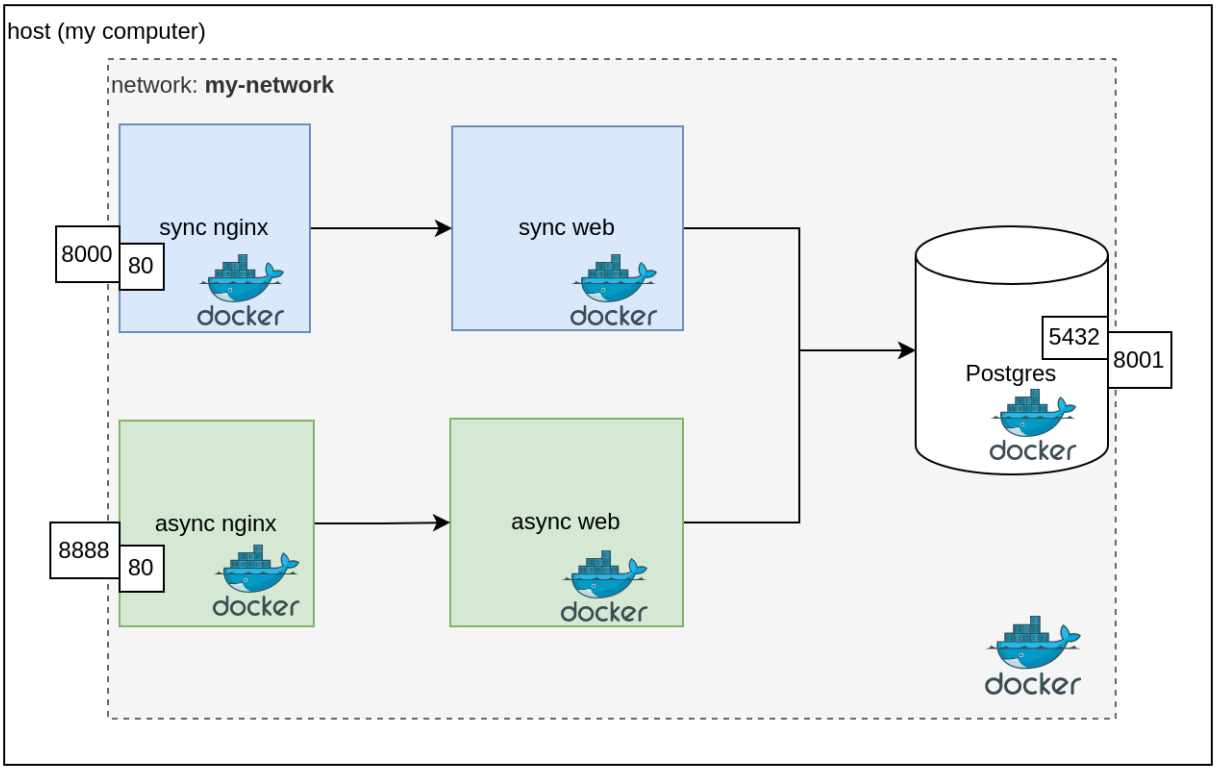
The demo project

I have uploaded the source code of the demo project here. The **mysite_async** folder contains the source code of the async build. The **mysite_sync** folder contains the source code of the sync build. We are going to use them in the rest of the article. They have the dependencies as follows:

- Python3.8
- Django3.2
- Docker
- Apache HTTP server benchmarking tool (aka "**ab**")
- GNU make. (The **Makefile** explains how we are going to build, run and test the builds.)

The **ab** and **GNU make** are optional. If you don't have it on your machine, no worries the demo should still work.

In a nutshell, the sync and async applications would run the same ORM query against a Postgres database table. Thanks to Docker, everything is containerized so that we can easily package the build. The architecture is as follows:



Implementation

Network

Create a network by the command as follows:

```
docker network create my-network
```

Database and ORM model

Let's launch a Postgresql database as a container. The **docker-compose.yml** is as follows:

```
version: "3.9"
services:
  db:
    image: postgres
    restart: always
    environment:
      POSTGRES_PASSWORD: mysecretpassword
    ports:
```

```
      - 8001:5432
    networks:
      - my-network

networks:
  my-network:
    external: true
```

Create a **polls_foo** table. It contains an **id** column. It is as follows:

id
1
2

Let's populate 1,000,000 rows by the shell command as follows:

```
#!/bin/sh

export PGPASSWORD=mysecretpassword

psql --host localhost --port 8001 --username=postgres -d postgres <<EOF
CREATE TABLE polls_foo AS
SELECT * FROM GENERATE_SERIES(1, 1000000) AS id;
EOF
```

The respective Django ORM model is as follows:

```
from django.db import models


class Foo(models.Model):
    id = models.IntegerField()

    class Meta:
        managed = False
```

Application

Let's create an app called **poll**. Then create a view that returns the count of **Foo** objects whose **id** is equal to or greater than 50. The code block of the sync view and function is as follows:

```
from polls.models import Foo
from django.http import HttpResponse


def index(request):
    count = Foo.objects.filter(id__gte=50).count()
    return HttpResponse(count)
```

The code block of the async view and function is almost the same. It's as follows:

```
from asgiref.sync import sync_to_async
from polls.models import Foo
from django.http import HttpResponse


def _find_count(value):
    return Foo.objects.filter(id__gte=value).count()


find_count = sync_to_async(_find_count, thread_sensitive=True)


async def index(request):
    count = await find_count(50)
    return HttpResponse(count)
```

We would like the view to be associated with endpoint **polls/**. Thus, the code is as follows:

```
from django.urls import path
from . import views

urlpatterns = [
    path('', views.index, name='index'),
]
```

Containerization

For the sync project, we are going to use uWSGI as the server and process manager. In a nutshell, the data flows like this: HTTP client ↔ Nginx ↔ uWSGI ↔ Django app.

For the async project, we are going to use Uvicorn as the server and Gunicorn as the process manager. In a nutshell, the data flows like this: HTTP client ↔ Nginx ↔ Gunicorn(Uvicorn) ↔ Django app.

In order to make testing fair, we specify each build only spawns one worker(process). The async setup in the **gunicorn.web.config.py** file is as follows:

```
wsgi_app = 'mysite_async.asgi:application'
worker_class = 'uvicorn.workers.UvicornWorker'
workers = 1
bind = 'unix:/tmp/gunicorn/gunicorn.sock'
accesslog = '-'
```

The sync setup in the **mysite.uwsgi.ini** file is as follows:

```
[uwsgi]

socket = /tmp/uwsgi/mysite.sock
module = mysite_sync.wsgi
master = true
processes = 1
listen = 500
chmod-socket = 666
vacuum = true
```

We probably want the two builds to run on separate ports locally. Thus, in the **docker-compose.yml** file, we make the sync project run on port 8000 while the async project runs on port 8888.

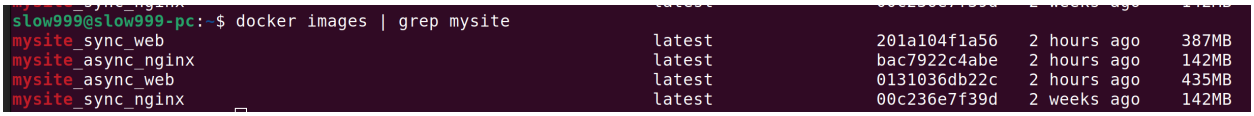
Regarding the rest of the settings, please check them in the git repository.

Build

Now, we are ready to build the Docker images. If your machine supports GNU make, you can simply run the command below under the root folder to build the images.

```
make build
```

We should expect that four images are created. They are **mysite_sync_web, mysite_sync_nginx, mysite_async_web,** and **mysite_async_nginx.** The snapshot is as follows:



If your machine does not support GNU make, you can build them in the following way.

Go to the **mysite_sync** folder. Then run commands as follows:

```
docker build -t mysite_sync_web .
docker build -f nginx/Dockerfile -t mysite_sync_nginx nginx
```

Go to the **mysite_async** folder. Then run commands as follows:

```
docker build -t mysite_async_web .
docker build -f nginx/Dockerfile -t mysite_async_nginx nginx
```

At this point, the Dockers images are ready. Let's run them in the next section.

Run

If your machine supports GNU make, you can simply run the two builds through one command under the root folder as follow:

```
make run
```

If your machine does not support GNU make, you can run the two builds separately by the following.

Go to the **mysite_sync** folder. Then run the command as follows:

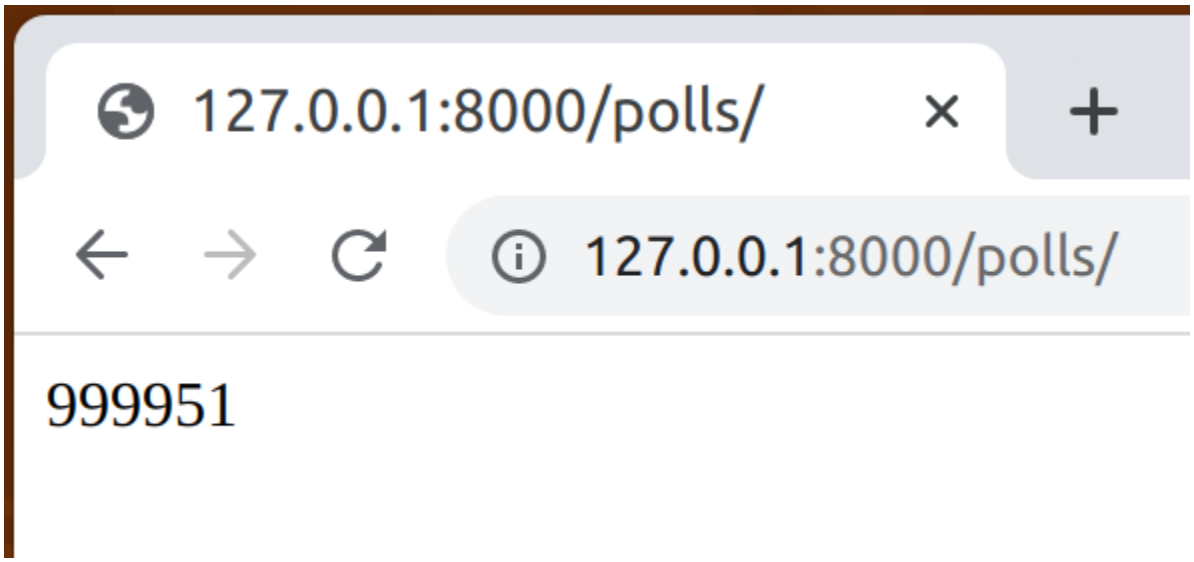
```
docker compose up -d
```

Go to the **mysite_async** folder. Then run the command as follows:

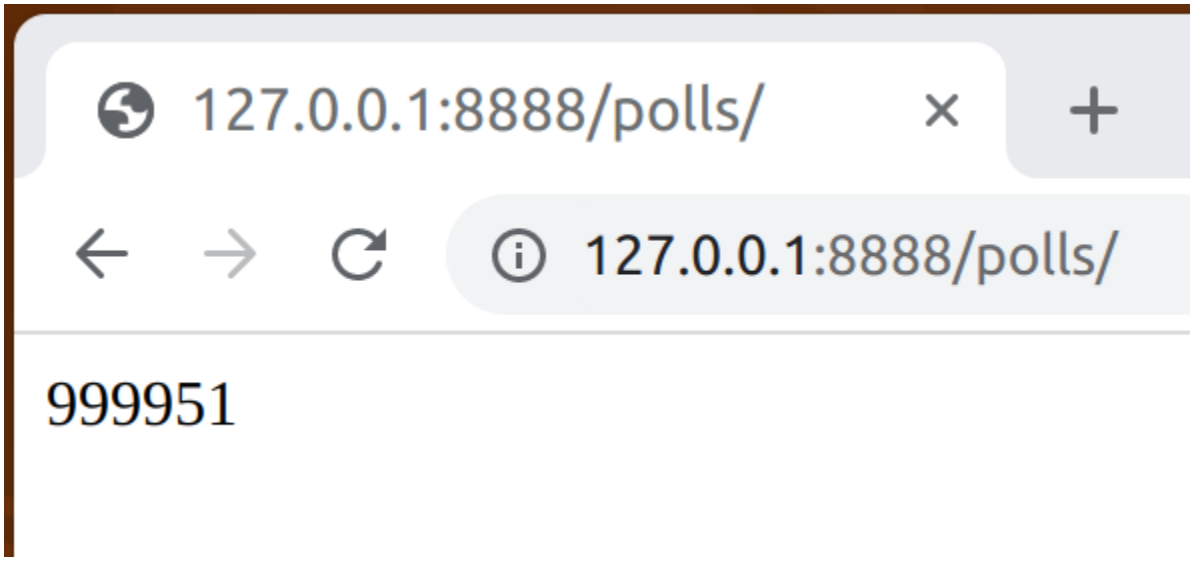
```
docker compose up -d
```

Again, note that the sync build should run on port 8000 while the async build should run on port 8888.

Checking the endpoint of the sync build, we should expect the snapshot as follow:



Checking the endpoint of the async build, we should expect the snapshot as follow. Since it does the same job as the sync build does back the scene, it should return 999951 as well.



Test

In this article, to simulate “busy” traffics, we are going to send 300 requests concurrently. Definitely, you can send more.

Let’s test the sync build. If your machine has Apache benchmarking tool installed, the command is as follows:

```
ab -n 300 -c 300 http://127.0.0.1:8000/polls/
```

Or if your machine support GNU, run the command as follows:

```
make test-sync
```

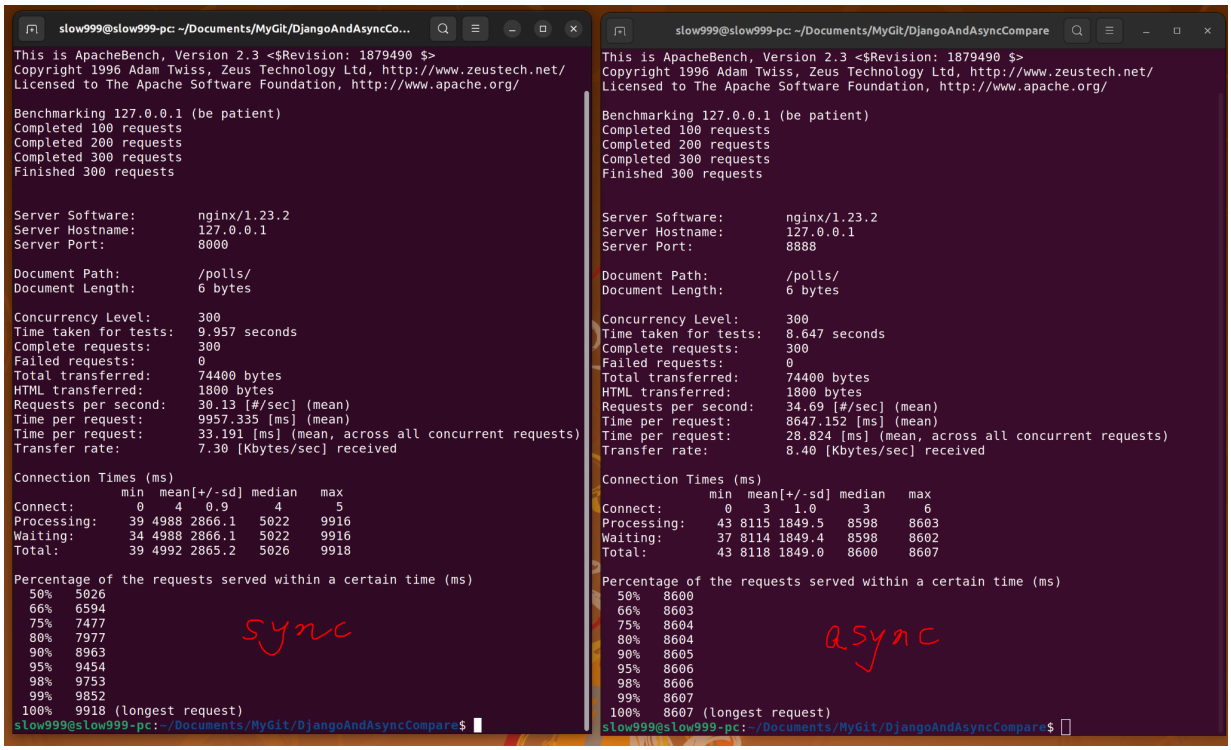
After the test is complete, we can test async build by using the command as follows:

```
ab -n 300 -c 300 http://127.0.0.1:8888/polls/
```

Or

```
make test-async
```

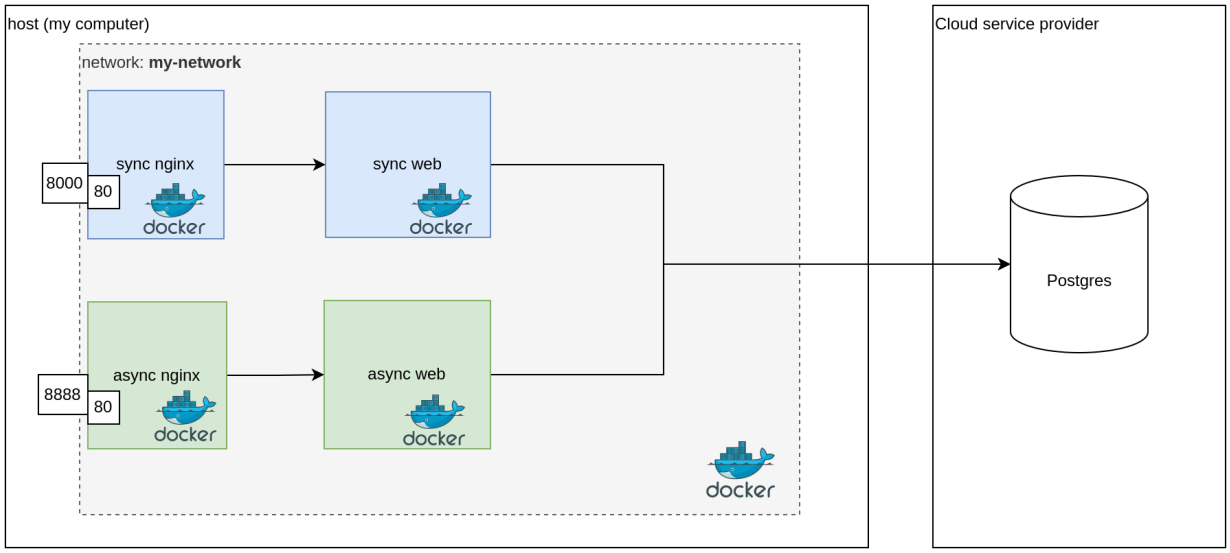
The snapshot of results is as follows:



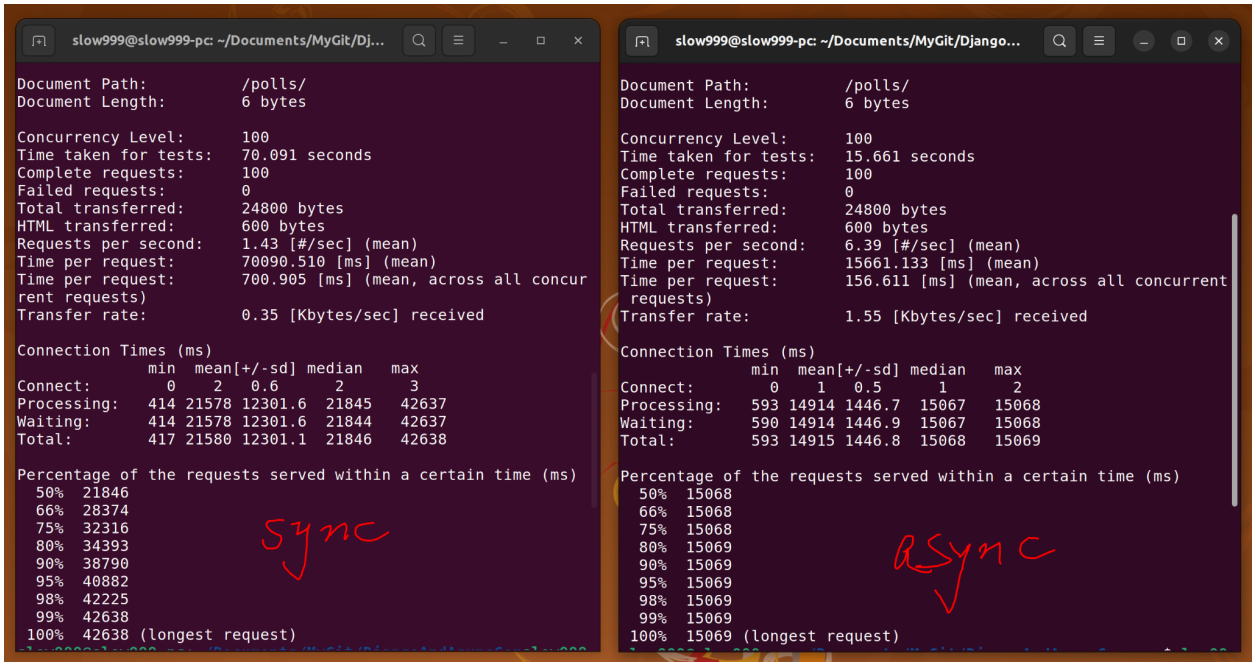
Result of sync build: It takes 9.957 s to complete the test. The fastest one returns in 39 ms. The most prolonged response takes 9918 ms to return. Half of the requests returns within 5026 ms. The response time seems to grow linear as more requests are handled.

Result of async build: It takes 8.647 s to complete the test. The fastest one returns in 43 ms. The most prolonged response takes 8607 ms to return. Half of the requests returns within 8600 ms. The response time seems constant as more requests are handled.

Regarding the total time, the async build has outperformed the sync build. The difference is not so obvious. Because in the testing of the async build, the Postgres Docker container might be overwhelmed (CPU-bound) when it handles 300 database queries at almost the same time. We can verify it by running another example in which I modified the database connection so that the builds connect to a remote Postgres database (I/O-bound) rather than a local Postgres Docker container. The architecture is as follows:



In this case, the database is, on earth, an “outsider”. The table they look up against is 17 GB and contains 6 million rows. The snapshot of results is as follows:



We can tell that the async build has significantly outperformed the sync build.

Sync or async? That depends.

- If the application would expect a small number of requests, there is no difference between the sync and async. In some cases, the sync build might even outperform.
- If the application would expect huge numbers of requests, you probably want to choose async.
- Use async build if the time it takes to complete a computation is determined by I/O-bound. It wouldn't benefit from writing the async view if a task is CPU-bound. For example, a function that finds all prime numbers between 0 and 1,000 is CPU bound. Thus, making it an async function doesn't improve performance that much. It could be worse due to managing the routine.

Conclusions

In this article, we went through the implementation of comparing sync and async Django views. We compared the performances in different scenarios. Actually, there is no right or wrong answer. Everything comes with a trade off. Please let me know what your thoughts are. In addition, if you are curious about the GNU make, please leave a comment. I will make a post regarding it. Stay tuned.

All source codes have been uploaded to the Git repository as follows:
https://github.com/slow999/DjangoAndAsyncCompare