Server Workers - Gunicorn with Uvicorn

Let's check back those deployment concepts from before:

- Security - HTTPS
- Running on startup
- Restarts
- **Replication (the number of processes running)**
- Memory
- Previous steps before starting

Up to this point, with all the tutorials in the docs, you have probably been running a **server program** like Uvicorn, running a **single process**.

When deploying applications you will probably want to have some **replication of processes** to take advantage of **multiple cores** and to be able to handle more requests.

As you saw in the previous chapter about Deployment Concepts{.internal-link target=_blank}, there are multiple strategies you can use.

Here I'll show you how to use **Gunicorn** with **Uvicorn worker processes**.

!!! info If you are using containers, for example with Docker or Kubernetes, I'll tell you more about that in the next chapter: FastAPI in Containers - Docker{.internal-link target=_blank}.

```
In particular, when running on **Kubernetes** you will probably **not** want to use Gunicorn and instead run **a single Uvicorn process per container**, but I'll tell you about it later in that chapter.
```

Gunicorn with Uvicorn Workers

Gunicorn is mainly an application server using the **WSGI standard**. That means that Gunicorn can serve applications like Flask and Django. Gunicorn by itself is not compatible with **FastAPI**, as FastAPI uses the newest **ASGI standard**.

But Gunicorn supports working as a **process manager** and allowing users to tell it which specific **worker process class** to use. Then Gunicorn would start one or more **worker processes** using that class.

And **Uvicorn** has a **Gunicorn-compatible worker class**.

Using that combination, Gunicorn would act as a **process manager**, listening on the **port** and the **IP**. And it would **transmit** the communication to the worker processes running the **Uvicorn class**.

And then the Gunicorn-compatible **Uvicorn worker** class would be in charge of converting the data sent by Gunicorn to the ASGI standard for FastAPI to use it.

Install Gunicorn and Uvicorn

```
$ pip install "uvicorn[standard]" gunicorn

---> 100%
```

That will install both Uvicorn with the `standard` extra packages (to get high performance) and Gunicorn.

Run Gunicorn with Uvicorn Workers

Then you can run Gunicorn with:

```
$ gunicorn main:app --workers 4 --worker-class uvicorn.workers.UvicornWorker --bind
0.0.0.0:80

[19499] [INFO] Starting gunicorn 20.1.0
[19499] [INFO] Listening at: http://0.0.0.0:80 (19499)
[19499] [INFO] Using worker: uvicorn.workers.UvicornWorker
[19511] [INFO] Booting worker with pid: 19511
[19513] [INFO] Booting worker with pid: 19513
[19514] [INFO] Booting worker with pid: 19514
[19515] [INFO] Booting worker with pid: 19515
[19511] [INFO] Started server process [19511]
[19511] [INFO] Waiting for application startup.
[19511] [INFO] Application startup complete.
[19513] [INFO] Started server process [19513]
[19513] [INFO] Waiting for application startup.
[19513] [INFO] Application startup complete.
[19514] [INFO] Started server process [19514]
[19514] [INFO] Waiting for application startup.
[19514] [INFO] Application startup complete.
[19515] [INFO] Started server process [19515]
[19515] [INFO] Waiting for application startup.
[19515] [INFO] Application startup complete.
```

Let's see what each of those options mean:

- `main:app` : This is the same syntax used by Uvicorn, `main` means the Python module named " `main` ", so, a file `main.py` . And `app` is the name of the variable that is the **FastAPI** application.
    - You can imagine that `main:app` is equivalent to a Python `import` statement like:

      ```
      from main import app
      ```

    - So, the colon in `main:app` would be equivalent to the Python `import` part in `from main import app` .
- `--workers` : The number of worker processes to use, each will run a Uvicorn worker, in this case, 4 workers.
- `--worker-class` : The Gunicorn-compatible worker class to use in the worker processes.
    - Here we pass the class that Gunicorn can import and use with:

      ```
      import uvicorn.workers.UvicornWorker
      ```

- `--bind` : This tells Gunicorn the IP and the port to listen to, using a colon (`:`) to separate the IP and the port.

- If you were running Uvicorn directly, instead of `--bind 0.0.0.0:80` (the Gunicorn option) you would use `--host 0.0.0.0` and `--port 80`.

In the output, you can see that it shows the **PID** (process ID) of each process (it's just a number).

You can see that:

- The Gunicorn **process manager** starts with PID `19499` (in your case it will be a different number).
- Then it starts `Listening at: http://0.0.0.0:80`.
- Then it detects that it has to use the worker class at `uvicorn.workers.UvicornWorker`.
- And then it starts **4 workers**, each with its own PID: `19511`, `19513`, `19514`, and `19515`.

Gunicorn would also take care of managing **dead processes** and **restarting** new ones if needed to keep the number of workers. So that helps in part with the **restart** concept from the list above.

Nevertheless, you would probably also want to have something outside making sure to **restart Gunicorn** if necessary, and also to **run it on startup**, etc.

Uvicorn with Workers

Uvicorn also has an option to start and run several **worker processes**.

Nevertheless, as of now, Uvicorn's capabilities for handling worker processes are more limited than Gunicorn's. So, if you want to have a process manager at this level (at the Python level), then it might be better to try with Gunicorn as the process manager.

In any case, you would run it like this:

```
$ uvicorn main:app --host 0.0.0.0 --port 8080 --workers 4
<font color="#A6E22E">INFO</font>:     Uvicorn running on <b>http://0.0.0.0:8080</b>
(Press CTRL+C to quit)
<font color="#A6E22E">INFO</font>:     Started parent process [<font
color="#A1EFE4"><b>27365</b></font>]
<font color="#A6E22E">INFO</font>:     Started server process [<font
color="#A1EFE4">27368</font>]
<font color="#A6E22E">INFO</font>:     Waiting for application startup.
<font color="#A6E22E">INFO</font>:     Application startup complete.
<font color="#A6E22E">INFO</font>:     Started server process [<font
color="#A1EFE4">27369</font>]
<font color="#A6E22E">INFO</font>:     Waiting for application startup.
<font color="#A6E22E">INFO</font>:     Application startup complete.
<font color="#A6E22E">INFO</font>:     Started server process [<font
color="#A1EFE4">27370</font>]
<font color="#A6E22E">INFO</font>:     Waiting for application startup.
<font color="#A6E22E">INFO</font>:     Application startup complete.
<font color="#A6E22E">INFO</font>:     Started server process [<font
color="#A1EFE4">27367</font>]
<font color="#A6E22E">INFO</font>:     Waiting for application startup.
<font color="#A6E22E">INFO</font>:     Application startup complete.
```

The only new option here is `--workers` telling Uvicorn to start 4 worker processes.

You can also see that it shows the **PID** of each process, `27365` for the parent process (this is the **process manager**) and one for each worker process: `27368`, `27369`, `27370`, and `27367`.

Deployment Concepts

Here you saw how to use **Gunicorn** (or Uvicorn) managing **Uvicorn worker processes** to **parallelize** the execution of the application, take advantage of **multiple cores** in the CPU, and be able to serve **more requests**.

From the list of deployment concepts from above, using workers would mainly help with the **replication** part, and a little bit with the **restarts**, but you still need to take care of the others:

- **Security - HTTPS**
- **Running on startup**
- ***Restarts***
- Replication (the number of processes running)
- **Memory**
- **Previous steps before starting**

Containers and Docker

In the next chapter about FastAPI in Containers - Docker{.internal-link target=_blank} I'll tell some strategies you could use to handle the other **deployment concepts**.

I'll also show you the **official Docker image** that includes **Gunicorn with Uvicorn workers** and some default configurations that can be useful for simple cases.

There I'll also show you how to **build your own image from scratch** to run a single Uvicorn process (without Gunicorn). It is a simple process and is probably what you would want to do when using a distributed container management system like **Kubernetes**.

Recap

You can use **Gunicorn** (or also Uvicorn) as a process manager with Uvicorn workers to take advantage of **multi-core CPUs**, to run **multiple processes in parallel**.

You could use these tools and ideas if you are setting up **your own deployment system** while taking care of the other deployment concepts yourself.

Check out the next chapter to learn about **FastAPI** with containers (e.g. Docker and Kubernetes). You will see that those tools have simple ways to solve the other **deployment concepts** as well. ✨