Deployments Concepts

When deploying a **FastAPI** application, or actually, any type of web API, there are several concepts that you probably care about, and using them you can find the **most appropriate** way to **deploy your application**.

Some of the important concepts are:

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

We'll see how they would affect **deployments**.

In the end, the ultimate objective is to be able to **serve your API clients** in a way that is **secure**, to **avoid disruptions**, and to use the **compute resources** (for example remote servers/virtual machines) as efficiently as possible. 🚀

I'll tell you a bit more about these **concepts** here, and that would hopefully give you the **intuition** you would need to decide how to deploy your API in very different environments, possibly even in **future** ones that don't exist yet.

By considering these concepts, you will be able to **evaluate and design** the best way to deploy **your own APIs**.

In the next chapters, I'll give you more **concrete recipes** to deploy FastAPI applications.

But for now, let's check these important **conceptual ideas**. These concepts also apply to any other type of web API. 💡

Security - HTTPS

In the previous chapter about HTTPS{.internal-link target=_blank} we learned about how HTTPS provides encryption for your API.

We also saw that HTTPS is normally provided by a component **external** to your application server, a **TLS Termination Proxy**.

And there has to be something in charge of **renewing the HTTPS certificates**, it could be the same component or it could be something different.

Example Tools for HTTPS

Some of the tools you could use as a TLS Termination Proxy are:

- Traefik
  - Automatically handles certificates renewals ✨
- Caddy
  - Automatically handles certificates renewals ✨
- Nginx
  - With an external component like Certbot for certificate renewals
- HAProxy
  - With an external component like Certbot for certificate renewals
- Kubernetes with an Ingress Controller like Nginx

    - With an external component like cert-manager for certificate renewals
- Handled internally by a cloud provider as part of their services (read below 👇)

Another option is that you could use a **cloud service** that does more of the work including setting up HTTPS. It could have some restrictions or charge you more, etc. But in that case, you wouldn't have to set up a TLS Termination Proxy yourself.

I'll show you some concrete examples in the next chapters.

Then the next concepts to consider are all about the program running your actual API (e.g. Uvicorn).

Program and Process

We will talk a lot about the running "**process**", so it's useful to have clarity about what it means, and what's the difference with the word "**program**".

What is a Program

The word **program** is commonly used to describe many things:

- The **code** that you write, the **Python files**.
- The **file** that can be **executed** by the operating system, for example: `python`, `python.exe` or `uvicorn`.
- A particular program while it is **running** on the operating system, using the CPU, and storing things on memory. This is also called a **process**.

What is a Process

The word **process** is normally used in a more specific way, only referring to the thing that is running in the operating system (like in the last point above):

- A particular program while it is **running** on the operating system.
    - This doesn't refer to the file, nor to the code, it refers **specifically** to the thing that is being **executed** and managed by the operating system.
- Any program, any code, **can only do things** when it is being **executed**. So, when there's a **process running**.
- The process can be **terminated** (or "killed") by you, or by the operating system. At that point, it stops running/being executed, and it can **no longer do things**.
- Each application that you have running on your computer has some process behind it, each running program, each window, etc. And there are normally many processes running **at the same time** while a computer is on.
- There can be **multiple processes** of the **same program** running at the same time.

If you check out the "task manager" or "system monitor" (or similar tools) in your operating system, you will be able to see many of those processes running.

And, for example, you will probably see that there are multiple processes running the same browser program (Firefox, Chrome, Edge, etc). They normally run one process per tab, plus some other extra processes.

Now that we know the difference between the terms **process** and **program**, let's continue talking about deployments.

Running on Startup

In most cases, when you create a web API, you want it to be **always running**, uninterrupted, so that your clients can always access it. This is of course, unless you have a specific reason why you want it to run only in certain situations, but most of the time you want it constantly running and **available**.

In a Remote Server

When you set up a remote server (a cloud server, a virtual machine, etc.) the simplest thing you can do is to run Uvicorn (or similar) manually, the same way you do when developing locally.

And it will work and will be useful **during development**.

But if your connection to the server is lost, the **running process** will probably die.

And if the server is restarted (for example after updates, or migrations from the cloud provider) you probably **won't notice it**. And because of that, you won't even know that you have to restart the process manually. So, your API will just stay dead. 😱

Run Automatically on Startup

In general, you will probably want the server program (e.g. Uvicorn) to be started automatically on server startup, and without needing any **human intervention**, to have a process always running with your API (e.g. Uvicorn running your FastAPI app).

Separate Program

To achieve this, you will normally have a **separate program** that would make sure your application is run on startup. And in many cases, it would also make sure other components or applications are also run, for example, a database.

Example Tools to Run at Startup

Some examples of the tools that can do this job are:

- Docker
- Kubernetes
- Docker Compose
- Docker in Swarm Mode
- Systemd
- Supervisor
- Handled internally by a cloud provider as part of their services
- Others...

I'll give you more concrete examples in the next chapters.

Restarts

Similar to making sure your application is run on startup, you probably also want to make sure it is **restarted** after failures.

We Make Mistakes

We, as humans, make **mistakes**, all the time. Software almost *always* has **bugs** hidden in different places. 🐛

And we as developers keep improving the code as we find those bugs and as we implement new features (possibly adding new bugs too 😅).

Small Errors Automatically Handled

When building web APIs with FastAPI, if there's an error in our code, FastAPI will normally contain it to the single request that triggered the error. 🛡

The client will get a **500 Internal Server Error** for that request, but the application will continue working for the next requests instead of just crashing completely.

Bigger Errors - Crashes

Nevertheless, there might be cases where we write some code that **crashes the entire application** making Uvicorn and Python crash. 💥

And still, you would probably not want the application to stay dead because there was an error in one place, you probably want it to **continue running** at least for the *path operations* that are not broken.

Restart After Crash

But in those cases with really bad errors that crash the running **process**, you would want an external component that is in charge of **restarting** the process, at least a couple of times...

!!! tip ...Although if the whole application is just **crashing immediately** it probably doesn't make sense to keep restarting it forever. But in those cases, you will probably notice it during development, or at least right after deployment.

```
So let's focus on the main cases, where it could crash entirely in some particular
cases **in the future**, and it still makes sense to restart it.
```

You would probably want to have the thing in charge of restarting your application as an **external component**, because by that point, the same application with Uvicorn and Python already crashed, so there's nothing in the same code of the same app that could do anything about it.

Example Tools to Restart Automatically

In most cases, the same tool that is used to **run the program on startup** is also used to handle automatic **restarts**.

For example, this could be handled by:

- Docker
- Kubernetes
- Docker Compose
- Docker in Swarm Mode
- Systemd
- Supervisor
- Handled internally by a cloud provider as part of their services
- Others...

Replication - Processes and Memory

With a FastAPI application, using a server program like Uvicorn, running it once in **one process** can serve multiple clients concurrently.

But in many cases, you will want to run several worker processes at the same time.

Multiple Processes - Workers

If you have more clients than what a single process can handle (for example if the virtual machine is not too big) and you have **multiple cores** in the server's CPU, then you could have **multiple processes** running with the same application at the same time, and distribute all the requests among them.

When you run **multiple processes** of the same API program, they are commonly called **workers**.

Worker Processes and Ports

Remember from the docs About HTTPS{.internal-link target=_blank} that only one process can be listening on one combination of port and IP address in a server?

This is still true.

So, to be able to have **multiple processes** at the same time, there has to be a **single process listening on a port** that then transmits the communication to each worker process in some way.

Memory per Process

Now, when the program loads things in memory, for example, a machine learning model in a variable, or the contents of a large file in a variable, all that **consumes a bit of the memory (RAM)** of the server.

And multiple processes normally **don't share any memory**. This means that each running process has its own things, variables, and memory. And if you are consuming a large amount of memory in your code, **each process** will consume an equivalent amount of memory.

Server Memory

For example, if your code loads a Machine Learning model with **1 GB in size**, when you run one process with your API, it will consume at least 1 GB of RAM. And if you start **4 processes** (4 workers), each will consume 1 GB of RAM. So in total, your API will consume **4 GB of RAM**.

And if your remote server or virtual machine only has 3 GB of RAM, trying to load more than 4 GB of RAM will cause problems. 🚨

Multiple Processes - An Example

In this example, there's a **Manager Process** that starts and controls two **Worker Processes**.

This Manager Process would probably be the one listening on the **port** in the IP. And it would transmit all the communication to the worker processes.

Those worker processes would be the ones running your application, they would perform the main computations to receive a **request** and return a **response**, and they would load anything you put in variables in RAM.

And of course, the same machine would probably have **other processes** running as well, apart from your application.

An interesting detail is that the percentage of the **CPU used** by each process can **vary** a lot over time, but the **memory (RAM)** normally stays more or less **stable**.

If you have an API that does a comparable amount of computations every time and you have a lot of clients, then the **CPU utilization** will probably *also be stable* (instead of constantly going up and down quickly).

Examples of Replication Tools and Strategies

There can be several approaches to achieve this, and I'll tell you more about specific strategies in the next chapters, for example when talking about Docker and containers.

The main constraint to consider is that there has to be a **single** component handling the **port** in the **public IP**. And then it has to have a way to **transmit** the communication to the replicated **processes/workers**.

Here are some possible combinations and strategies:

- **Gunicorn** managing **Uvicorn workers**
    - Gunicorn would be the **process manager** listening on the **IP** and **port**, the replication would be by having **multiple Uvicorn worker processes**
- **Uvicorn** managing **Uvicorn workers**
    - One Uvicorn **process manager** would listen on the **IP** and **port**, and it would start **multiple Uvicorn worker processes**
- **Kubernetes** and other distributed **container systems**
    - Something in the **Kubernetes** layer would listen on the **IP** and **port**. The replication would be by having **multiple containers**, each with **one Uvicorn process** running
- **Cloud services** that handle this for your
    - The cloud service will probably **handle replication for you**. It would possibly let you define **a process to run**, or a **container image** to use, in any case, it would most probably be **a single Uvicorn process**, and the cloud service would be in charge of replicating it.

!!! tip Don't worry if some of these items about **containers**, Docker, or Kubernetes don't make a lot of sense yet.

```
I'll tell you more about container images, Docker, Kubernetes, etc. in a future
chapter: [FastAPI in Containers - Docker](./docker.md){.internal-link target=_blank}.
```

Previous Steps Before Starting

There are many cases where you want to perform some steps **before starting** your application.

For example, you might want to run **database migrations**.

But in most cases, you will want to perform these steps only **once**.

So, you will want to have a **single process** to perform those **previous steps**, before starting the application.

And you will have to make sure that it's a single process running those previous steps *even* if afterwards, you start **multiple processes** (multiple workers) for the application itself. If those steps were run by **multiple processes**, they would **duplicate** the work by running it on **parallel**, and if the steps were something delicate like a database migration, they could cause conflicts with each other.

Of course, there are some cases where there's no problem in running the previous steps multiple times, in that case, it's a lot easier to handle.

!!! tip Also, have in mind that depending on your setup, in some cases you **might not even need any previous steps** before starting your application.

```
In that case, you wouldn't have to worry about any of this. 🤷
```

Examples of Previous Steps Strategies

This will **depend heavily** on the way you **deploy your system**, and it would probably be connected to the way you start programs, handling restarts, etc.

Here are some possible ideas:

- An "Init Container" in Kubernetes that runs before your app container
- A bash script that runs the previous steps and then starts your application
  - You would still need a way to start/restart *that* bash script, detect errors, etc.

!!! tip I'll give you more concrete examples for doing this with containers in a future chapter: FastAPI in Containers - Docker{.internal-link target=_blank}.

Resource Utilization

Your server(s) is (are) a **resource**, you can consume or **utilize**, with your programs, the computation time on the CPUs, and the RAM memory available.

How much of the system resources do you want to be consuming/utilizing? It might be easy to think "not much", but in reality, you will probably want to consume **as much as possible without crashing**.

If you are paying for 3 servers but you are using only a little bit of their RAM and CPU, you are probably **wasting money** 💸, and probably **wasting server electric power** 🌎, etc.

In that case, it could be better to have only 2 servers and use a higher percentage of their resources (CPU, memory, disk, network bandwidth, etc).

On the other hand, if you have 2 servers and you are using **100% of their CPU and RAM**, at some point one process will ask for more memory, and the server will have to use the disk as "memory" (which can be thousands of times slower), or even **crash**. Or one process might need to do some computation and would have to wait until the CPU is free again.

In this case, it would be better to get **one extra server** and run some processes on it so that they all have **enough RAM and CPU time**.

There's also the chance that for some reason you have a **spike** of usage of your API. Maybe it went viral, or maybe some other services or bots start using it. And you might want to have extra resources to be safe in those cases.

You could put an **arbitrary number** to target, for example, something **between 50% to 90%** of resource utilization. The point is that those are probably the main things you will want to measure and use to tweak your deployments.

You can use simple tools like `htop` to see the CPU and RAM used in your server or the amount used by each process. Or you can use more complex monitoring tools, which may be distributed across servers, etc.

Recap

You have been reading here some of the main concepts that you would probably need to have in mind when deciding how to deploy your application:

- Security - HTTPS
- Running on startup
- Restarts

- Replication (the number of processes running)
- Memory
- Previous steps before starting

Understanding these ideas and how to apply them should give you the intuition necessary to take any decisions when configuring and tweaking your deployments. 😳

In the next sections, I'll give you more concrete examples of possible strategies you can follow. 🚀