# FastAPI from a CRUDmaker's POV: A Start-to-Finish Example

Daniil Kraynov – parkanaur.net

February 25, 2021

# Contents

# 1   Introduction

FastAPI, a high-performant Python web framework, is very well documented, but the documentation might be a bit too overwhelming for a semi-seasoned developer who already has experience with designing web applications and has a general stance on how they should be structured.

I've started writing this article as a compilation of notes to refer to when writing an app in FastAPI from the ground up. The end result of my initial tinkering with FastAPI, documented here, is a file hosting application with a bunch of extras like Docker integration.

Due to FastAPI being an ASGI framework, deployment process is a little bit different from using nginx and uWSGI, so I hope to cover the deployment caveats as well.

## 1.1   Who is this article intended for

This article aims to help developers coming from other frameworks (Flask, Django, as well as non-Python ones) who know what they're doing and are willing to jump straight into action but don't feel like getting overwhelmed with deeper aspects of FastAPI's official documentation [1] (I still advise everyone to read it - it's good on its own and is quite helpful!).

It could also be of good use to beginner developers, but some aspects of this article *might* be harder to understand for them - please give feedback on whether it's a good read or not!

## 1.2   Application building steps

From my experience, the general workflow for producing a mid-sized web application is as follows:

1. Designing a set of domain model classes (e.g. User, Item, etc.)
   (a) Choosing a data source
   (b) Likely some automatic migration scripts
2. Creating an API skeleton for at least one entity
   (a) Authentication logic
3. Adding a business logic layer for at least one entity
4. Binding it all together
5. Adding logic/API for the rest of the entities
6. [optional] Adding a frontend and a bunch of non-logic pages (in parallel with the previous steps or after them, depending on size of the developing team)
7. [optional] Deploy scripts, tests, CI/CD, etc.

FastAPI's documentation describes or mentions most of these steps, but in no specific order. For an experienced developer there are pages of particular interest (e.g. app structure for bigger applications [2], but from there you have to go deeper into the documentation in order to find out how to structure your models, DTOs, etc.

You'd probably start jumping around more and more around the documentation, get overwhelmed with the amount of points (most likely you wouldn't need all of them at the start of your FastAPI journey), and, in the worst case, lose motivation in learning FastAPI - it happened to me at first, which is a shame, because FastAPI is a beautiful framework!

## 1.3   Feedback

I don't consider myself an expert in either programming, documentation, or English, and this article is certain to have bad architecture decisions or mishaps in general.

---

[1] https://fastapi.tiangolo.com
[2] https://fastapi.tiangolo.com/tutorial/bigger-applications/

I encourage you to send your comments regarding this article, whether they're about me doing a good job or being an ignorant fool who doesn't know any better (I'm serious on this one!). Maybe you can think of a few additions as well, in which case you can also fork this article - it's "licensed" under CC0, so you're free to do whatever you want with it.

You can send your comments via GitHub issues/PRs [3] or by sending an e-mail at dan @ parkanaur.net. Don't hesitate!

---

[3]https://github.com/parkanaur/fastapi-notes

# 2 Preparations

The project that I'm going to describe is **PyFH** [4], a file hosting application designed for self-hosting, with a few additional features like public/private files, timed files, encryption, etc.

I'll try to follow the incremental model, starting at the very basic things like setting up environment all the way to deployment on a real world VPS.

## 2.1 Setting up environment

It seems that the good old requirements.txt is slowly falling out of favor within the Python ecosphere, so we're gonna ~~bite the bullet~~ follow suit and use the newer toolset (i.e. Poetry). This is a good exercise in using non-standard tooling as well.

Poetry is supposed to make managing virtualenvs and dependencies easier, so let's go ahead and install it [5].

You can use regular requirements.txt instead - be sure to remove Poetry-related lines and replace them with manual venv creation (`python -m venv venv`).

```
1   # Poetry will create the project directory for you automatically
2   cd ~/code
3   poetry new pyfh
4   cd pyfh
5   git init
6
7   # Virtualenv creation/activation (deactivate with exit)
8   poetry shell
9
10  # Don't @ me
11  mv README.rst README.md
```

Time to add dependencies. We're building a file hosting app, so we're gonna need a few extras for our FastAPI dependency. It's a good idea to just add the whole `fastapi[all]` dependency during development - it adds multipart support, as well as a few more goodies, and if you care about the amount of your dependencies, you can clean the unneeded stuff out later.

`fastapi[all]` also installs `uvicorn[standard]` as per FastAPI installation tutorial.

**Note:** Most of the bash commands will imply that you have activated your virtualenv (`poetry shell`).

```
1   # Install our dependency - same as (pip install + freeze > requirements.txt)
2   # If you're using zsh, enclose fastapi[all] in quotes
3   poetry add fastapi[all]
```

Now we're ready to start adding more stuff into the project.

## 2.2 Initial project structure

First we have to add some boring stuff - a few folders with empty `__init.py__`'s in them, a barebones main.py file, .gitignore, etc.

### 2.2.1 `main.py`

This is similar to `main.py/wsgi.py` in Flask - an ASGI server will use it as a starting point for your application (same as passing `main:app` to uWSGI, if you will).

Let's add the simplest `main.py` possible for now:

---

[4]https://github.com/parkanaur/pyfh
[5]https://python-poetry.org/docs/

```
1   from fastapi import FastAPI
2
3   app = FastAPI()
4
5   @app.get("/")
6   async def test():
7       return "Welcome to pyfh!"
```

Note the `async` keyword - it's used extensively in FastAPI. Asynchronous programming is a topic which isn't covered extensively in this article, but what you should know is that in most of the cases it really speeds things up. It is native to FastAPI development, in particular to path operations (`fastapi.get(), .post()`, etc.) - read more at the official documentation [6].

Let's try the code out. Uvicorn is used to launch ASGI apps [7] (we'll cover production deployment later), `--reload` enables auto-restart on code changes:

```
1   uvicorn main:app --reload
2   # ..... ^^^^---- main.py
3   # ..... -----^^^ app = FastAPI()
```

Alternatively you can call `uvicorn.run()` in main.py, rather than use the command-line tool.

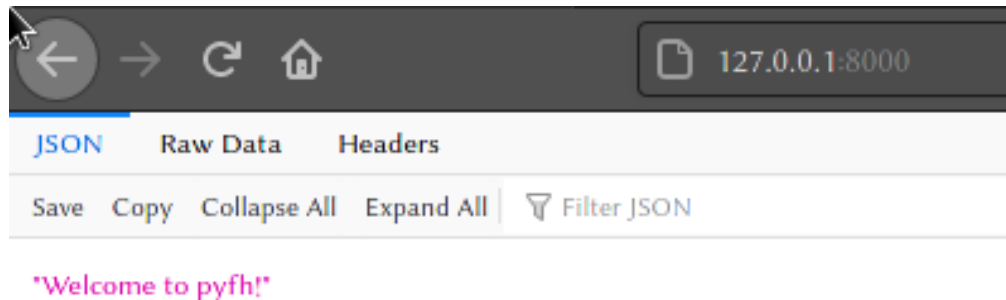Then make an HTTP request to 127.0.0.1:8000 and you should see something along the lines of the following:



Figure 1: Hello World in FastAPI

### 2.2.2 Folders and `__init.py__`'s

Time to add some more structure. Files not in the root directory **are empty** unless otherwise noted. To save time, I'll assume you know what `__init.py__`'s are for.

- ~/code/pyfh
    - main.py
    - pyproject.toml
    - poetry.lock
    - .gitignore
    - README.md
    - [optional] .env
    - Dockerfile *
    - .dockerignore *
    - docker-compose.yml *

---

[6] https://fastapi.tiangolo.com/async/
[7] https://www.uvicorn.org/deployment/

- pyfh/
    * __init.py__
    * config.py
    * entities/
        · __init.py__
        · user.py
    * routes/
        · __init.py__
        · user.py

\* You may have noticed the presence of Docker-related files. We'll put Docker to good use in the next section. As much as I hate to admit it, Docker's just too useful sometimes.

### 2.2.3 gitignore

Github's Python gitignore will do just fine [8], but read the notes below.

**Note:** You don't need to add `poetry.lock` to .gitignore - it's used to freeze currently installed (on your machine) package versions (similar to versions in `requirements.txt`). Poetry will make use of that file on other developers' machines to produce the same environment you have. You can later run `poetry update` to use newer package versions.

**Note:** Be sure to add `.env` to your .gitignore if you follow the structure above. We'll only use that .env file for development purposes, and it's generally **not recommended** to keep configuration data inside your codebase. When preparing your code for production deployment, make sure to keep .env files in a secure location.

## 2.3 Configuration files

I firmly believe that for a project with more than a few files of code it is important that you deal with configuration and deployment details as early as possible.

Setting up proper configuration for your project can be hard, especially if you want it to be easy to use in production later on and have little experience in starting projects from scratch.

### 2.3.1 Environment variables

It is common practice to use **environment variables** as configuration source for applications. They're used extensively in Docker and Docker Compose too.

In Linux, env variables are set the following way:

```
1  export VAR_NAME="var_value"
2  export VAR_INT_VALUE=12
```

In Python you read them the following way:

```
1  import os
2  # default value argument is optional
3  var_name = os.getenv("VAR_NAME", "default value") # -> "var_value"
4  var_int_value = os.getenv("VAR_INT_VALUE") # -> "12" <- note that os.getenv always returns a string
```

It is also common to write **env files** (prefixed with dot: **.env**) to store environment variables in. You may have noticed files that look like this when working with Docker images:

File 1: `.env` (example)

---

[8]https://github.com/github/gitignore/blob/master/Python.gitignore

```
1      VAR_NAME=var_value
2      VAR_INT_VALUE=12
```

### 2.3.2 Managing configuration in FastAPI with Pydantic

You could make use of `os.getenv`, but FastAPI docs suggest a different way, and it's much more elegant.

FastAPI utilizes at its core Pydantic, a data validation library, to check what's coming from users. It is also used for settings management [9], and it makes working with configurations a breeze.

We have created a separate **config.py** for managing project configuration - let's use it. Right now we only need database info - the rest will be added as needed.

File 2: `pyfh/config.py`

```
1   from pydantic import BaseSettings
2
3
4   class Settings(BaseSettings):
5       db_host: str = "127.0.0.1"
6       db_port: int = 5432
7
8       # Nested "Config" classes are used by Pydantic classes for additional base classes configuration,
9       # e.g. env file location for BaseSettings / orm mode usage for Pydantic models (BaseModel)
10      class Config:
11          env_file = ".env"
12          #case_sensitive = False
13
14  settings = Settings()
```

All required environment variables are then read by Pydantic when `Settings` is initialized as a variable on line 14.

**Note:** Pydantic reads environment variables in a **case-insensitive** way. In the following file, both variables are eligible to be read into the class above:

File 3: `.env`

```
1   db_host=10.0.12.23
2   DB_PORT=5000
```

You can change that behavior by uncommenting the line in the `Settings.Config` class.

The order in which Pydantic reads environment variables is as follows:

1. If defined, exported (not in the **.env**) variables are used
2. If defined, the **.env** variables are used
3. Fallback to default values (127.0.0.1 and 5432 in our case)

That means `export DB_HOST=10.0.0.1` will take precedence over `DB_HOST=127.0.0.1` from **.env**.

**Note:** If you're using development and/or deployment via Docker/Docker Compose, you most probably **don't** need to use env file configuration in a Pydantic class - in this case environment variables are defined and passed to the application using one of the following means:

- `docker` run -e VAR_NAME=var_value
- `docker` run --env-file filename
- env_file field in docker-compose.yml

---
[9]https://pydantic-docs.helpmanual.io/usage/settings/

If you still have questions, read up on official FastAPI docs for settings management, which should answer them all [10].

## 2.4   Docker configuration

In layman's terms, a Docker image is a packaged environment (like virtualenv) which contains your project and which can be configured, deployed and run in one command on any Linux machine which has Docker server installed.

While you certainly can make do without Docker, integrating it into your workflow could save a lot of time during development of deployment routines and during deployment itself.

### 2.4.1   `Dockerfile`

Dockerfile dictates how a Docker image for the project is built.

One option would be writing your image from scratch (i.e. by starting with an OS or Python image: `FROM ubuntu:18.04` / `FROM python:3.9.2-slim`), in which case you'll have to do some more work, like adding Python or web servers (nginx, gunicorn, etc), yourself.

Another option would be using more advanced images which include most of the needed tools. FastAPI's author has provided developers with a few useful images with support for all tools required for proper FastAPI production deployment. The one we're going to use in this guide is `tiangolo/uvicorn-gunicorn-fastapi` [11].

That image automatically includes and sets up Uvicorn and Gunicorn (more on that later), and actually launches your application from a fixed Python file location (i.e. you create the file in that location and the image reads that file during deployment).

Here's the template Dockerfile [12], which we'll get to editing in a moment.

File 4: `Dockerfile`

```
1   FROM tiangolo/uvicorn-gunicorn-fastapi:python3.7
2
3   # Install Poetry
4   RUN curl -sSL https://raw.githubusercontent.com/python-poetry/poetry/master/get-poetry.py | POETRY_HOME=/opt/
        poetry python && \
5       cd /usr/local/bin && \
6       ln -s /opt/poetry/bin/poetry && \
7       poetry config virtualenvs.create false
8
9   # Copy using poetry.lock* in case it doesn't exist yet
10  COPY ./app/pyproject.toml ./app/poetry.lock* /app/
11
12  RUN poetry install --no-root --no-dev
13
14  COPY ./app /app
```

**Suiting the image to our needs**

The image's README [13] suggests placing our `main.py` file (and, consequently, the rest of the source code) into `app` directory - the `app/main.py` is where the image looks for an `app` variable (i.e. FastAPI object - similar to passing `main:app` to uWSGI).

It is quite likely that you will want to structure your code as a Python package and have different naming applied to it. In our case, the source code is contained in the `pyfh` folder, not `app`. So, there are two ways to circumvent the requirement described above.

---

[10]https://fastapi.tiangolo.com/advanced/settings
[11]https://github.com/tiangolo/uvicorn-gunicorn-fastapi-docker
[12]https://github.com/tiangolo/uvicorn-gunicorn-fastapi-docker#using-poetry
[13]https://github.com/tiangolo/uvicorn-gunicorn-fastapi-docker#how-to-use

- In Dockerfile, manually map the app structure to an image conformant one:

```
1  COPY ./pyproject.toml ./poetry.lock* /app/
2  COPY ./pyfh /app
```

This method still has the drawback of having to conform to variable naming and is more error-prone in general, since you now have to keep track of in-image project structure, as well as adjust the rest of the project (tests, CI, etc.)

- Use environment variables via `docker run -e` (see Docker environment variables passing in Section 2.3.2) or Docker Compose. In the image we're using, you can pass your new main file location (via the `MODULE_NAME` variable), FastAPI app variable name (`VARIABLE_NAME`), and so on [14]. We'll use that option when writing a Docker Compose file (Section 2.4.2)

### 2.4.2 `docker-compose.yml`

Docker Compose allows you to define, execute, and manage a single-/multicontainer application using YAML files. In our case, apart from the FastAPI application, we're going to be using a PostgreSQL database (in its own conatiner) and a frontend application (we'll worry about that later). Let's look at our `docker-compose.yml`:

### 2.4.3 `.dockerignore`

Dockerignore is to Docker what gitignore is to git. You don't want to include intermediate files in your image to save space, and you don't want sensitive data left behind as well. The rule of thumb is that you include everything you'd include in `.gitignore` plus a few more additions to reduce image size (e.g. `.git` folder).

## 2.5 Technical details

### 2.5.1 On Gunicorn, Uvicorn and Nginx integration

### 2.5.2 Nginx: inside or outside of Docker?

### 2.5.3 Docker cache invalidation and command order

# 3 Domain Model

## 3.1 Creating entities

---

[14]https://github.com/tiangolo/uvicorn-gunicorn-fastapi-docker#advanced-usage