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

Daniil Kraynov – parkanaur.net

April 28, 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`).

```
# Poetry will create the project directory for you automatically
cd ~/code
poetry new pyfh
cd pyfh
git init

# Virtualenv creation/activation (deactivate with exit)
poetry shell

# Don't @ me
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`).

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

```python
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
async def test():
    return "Welcome to pyfh!"
```

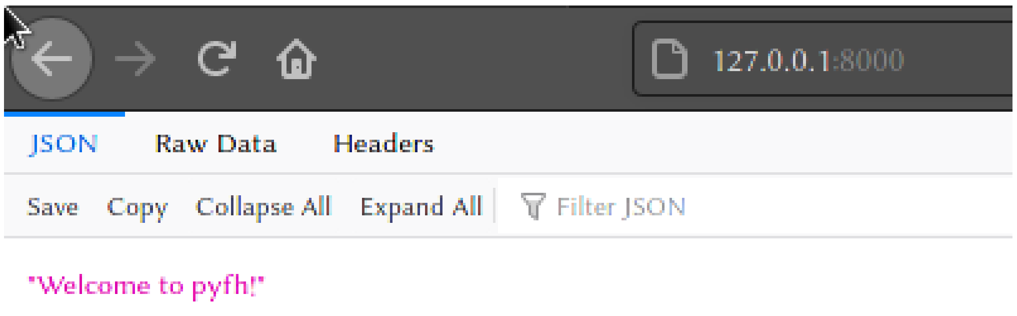
Note the `async` keyword - it's used extensively in FastAPI. Asynchronous programming is a topic which isn't covered extensively in this article, but what you should know is that in most of the cases it really speeds things up. It is native to FastAPI development, in particular to path operations (`fastapi.get()`,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

```
uvicorn main:app --reload
# ..... ^^^^---- main.py
# ..... -----^^^ 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:



```
"Welcome to pyfh!"
```

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 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:

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

In Python you read them the following way:

```
import os
# default value argument is optional
var_name = os.getenv("VAR_NAME", "default value") # -> "var_value"
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

```
VAR_NAME=var_value
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`

```
db_host=10.0.12.23
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 Docker commands - see Section 2.4.1

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

[9] https://pydantic-docs.helpmanual.io/usage/settings/
[10] https://fastapi.tiangolo.com/advanced/settings

2.4 Docker configuration

In layman's terms, a Docker image is a packaged environment (like virtualenv, but it encapsulates the whole operating system) 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 Passing environment variables to Docker containers

You can pass environment variables to Docker containers during startup:

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

2.4.2 `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], adjusted for our project.

File 4: `Dockerfile`

```
FROM tiangolo/uvicorn-gunicorn-fastapi:python3.8-slim

# slim image does not include curl - a small price to pay for extra 400mb of free space
RUN apt-get update && apt-get install -y --no-install-recommends \
    curl \
    && rm -rf /var/lib/apt/lists/*

# Install Poetry
RUN curl -sSL https://raw.githubusercontent.com/python-poetry/poetry/master/get-poetry.py | POETRY_HOME=/opt/
     poetry python && \
    cd /usr/local/bin && \
    ln -s /opt/poetry/bin/poetry && \
    poetry config virtualenvs.create false

# Copy using poetry.lock* in case it doesn't exist yet
COPY pyproject.toml poetry.lock* /app/

RUN poetry install --no-root --no-dev

```

[11] https://github.com/tiangolo/uvicorn-gunicorn-fastapi-docker
[12] https://github.com/tiangolo/uvicorn-gunicorn-fastapi-docker#using-poetry

```
COPY . /app
```

Suiting the image to our needs

Note: If you're writing your Dockerfile from the ground up, without using another image as a starting point, you don't need to read this subsection.

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).

You can control where the image will read your FastAPI object from by using 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 2.4.3)

Example (assume your `main.py` has been renamed to `other_main_file.py` and your `app = FastAPI()` has been replaced with `api = FastAPI()`):

```
    docker run -e APP_MODULE="other_main_file:api" image
```

Note: The image creates and uses as a working directory the `/app` directory (i.e. a folder on the same level as `/home`, `/var`, `/usr` and so on) - this might confuse you the first time you're reading its documentation. This picture should clear up any questions on how project-to-Docker-image mapping operates:

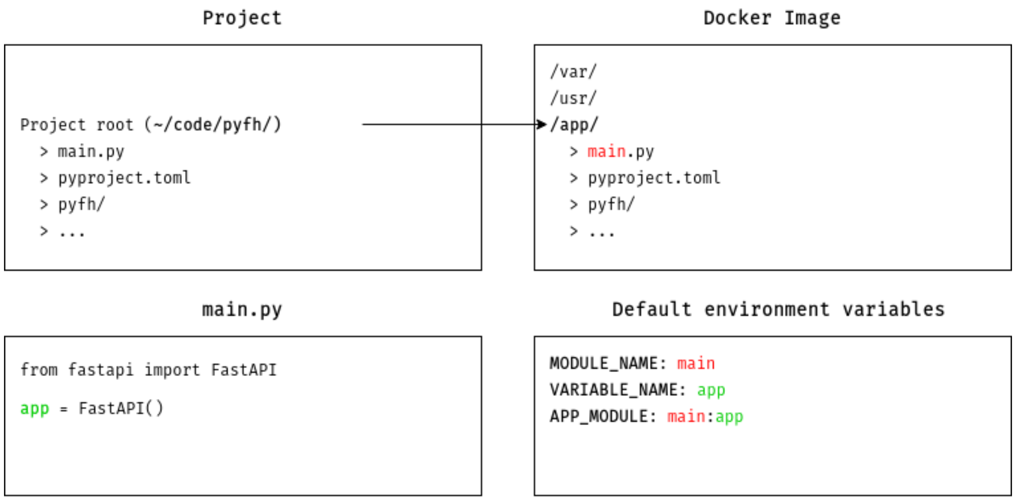


Figure 2: File mapping and environment variables

The `APP_MODULE` variable is actually used by Gunicorn to launch the app. It is either deduced from `MODULE_NAME` and `VARIABLE_NAME` or overridden by the user.

2.4.3 `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`:

```
version: "3.8"

services:
```

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

```
  backend:
    build: .
    depends_on:
      - database
    env_file:
      - .env
    ports:
      - "8000:8000"

  database:
    image: postgres:12
    env_file:
      - .env
    ports:
      - "5432:5432"
    volumes:
      - pg_data:/var/lib/postgresql/data

volumes:
  pg_data:
```

`services` is a collection of containers your app will have.

`build` is how to build the container image - i.e. in which folder your `Dockerfile` is located.

`depends_on` defines which container/s has/ve to be created **before** this container is created. We need database set up before the backend API is initialized - can't send requests to a null database.

`env_file` passes the environment variables from the passed file to the container.

Docker `ports` and `expose`

`ports` exposes selected container ports **to host machine, not the other containers** - the command to expose to other containers in the network is `EXPOSE` in `Dockerfile` or `expose` in `docker-compose.yml`. `ports` are defined in a few ways (more in official Docker documentation [15]):

1. `"port1:port2"`
   Here, `port1` is the port on which this container's app is exposed **to the host machine.**
   `port2` is the port **in the container** on which your application is running on.
   Assume the port mapping `"8000:80"`, as described in Figure 3.

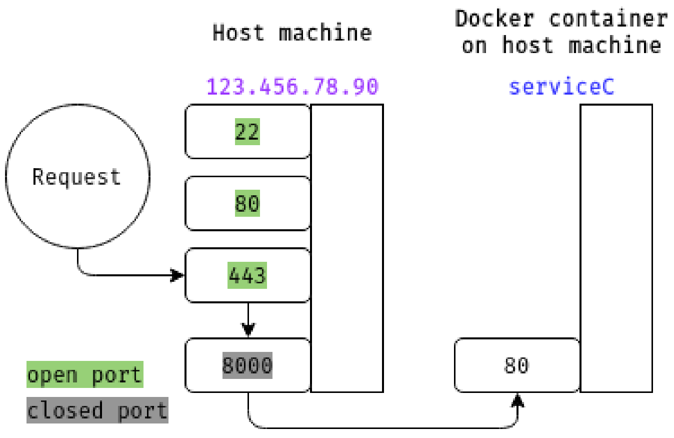


Figure 3: Example ports setup

[15]https://docs.docker.com/compose/compose-file/compose-file-v3/#ports

The port 8000 on machine 123.456.78.90 then redirects all requests to whatever is running on port 80 on Docker container serviceC.

2. "port1"
   This setting makes the port port1 available on the host machine via a **random** (ephemeral) unused port and is used to avoid port collisions. You can then find out what port on the host machine maps to container port via netstat or some other tool.

For an example usage of EXPOSE, you can refer to Dockerfile for the postgres image [16]. Whenever you're using that image, you don't need to expose the 5432 port, as it's already exposed as per the Dockerfile instruction (EXPOSE 5432). 5432 is accessible by default from other containers in the network, but you may also want to be able to access the database from the host machine - in which case you'll have to use ports.

Docker volumes

volumes are used to:

- **persist data between container recreations.** Whenever a container is recreated, its "writable layer" (a.k.a. container filesystem after the last Dockerfile instruction is executed) is destroyed and created anew, restoring all data (e.g. filesystem files - that includes database contents) to image defaults, as if you've just created a new container. Volumes are created and persisted **on the host machine** in a special place (/var/lib/docker/volumes by default). After the container is (re)created, these volumes are mounted inside the container. How and where they are mounted is specified by Docker/Docker Compose instructions:

File 5: Volume definitions in Docker Compose file (see Figure 4)

```
...

services:
    ...
    serviceC:
        ...
        volumes:
            - pg_data:/var/lib/postgresql/data
            # technically this is a bind mount - a directory from host machine is mounted inside
                 the container
            - /home/user/some_folder:/opt/more_data

volumes:
    # Docker files are created automatically for this volume in /var/lib/docker/volumes
    pg_data:
```

[16] https://github.com/docker-library/postgres/blob/master/Dockerfile-alpine.template#L182

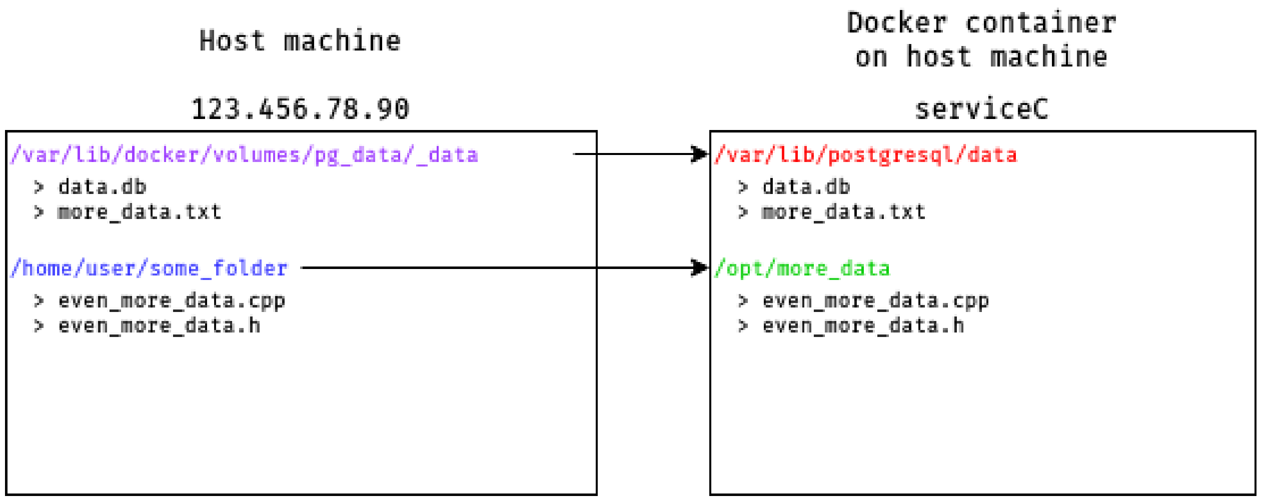


Figure 4: How volume mounting works for named volumes and host-mounted directories

- **share data between multiple containers.** Obviously, the same volume can be mounted on multiple containers in the network.

```
...

services:
    serviceA:
        ...
        volumes:
            - some_volume:/opt/data
    serviceB:
        ...
        volumes:
            - some_volume:/var/diff_data
    ...
...
volumes:
    some_volume:
```

You can read more about volumes in the official documentation [17] [18].

2.4.4 `.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

It might seem too much to have the actual app hidden behind multiple middlemen, but there are valid reasons to use mentioned tools:

[17] https://docs.docker.com/storage/
[18] https://docs.docker.com/compose/compose-file/compose-file-v3/#volumes

- **Uvicorn.** This actually launches your app the way you would launch it via `python script.py`, but with included asynchronicity.
- **Gunicorn.** It manages the number of uvicorn workers for added resilience and effectiveness.
- **Nginx.** Might not be necessary if you're running your setup on something like AWS where reverse proxying is handled automatically, but otherwise it manages incoming requests and allows to route them to multiple apps, if you have them. It also handles static files better than other instruments could.

2.5.2 Containerized nginx

There is an official Docker image for nginx [19] and you can use it for a fully containerized setup, but if you're running more than one web application on your own server, I'd recommend leaving nginx outside Docker - otherwise you risk having headaches from having to configure nginx to handle multiple Docker networks for different applications. From my experience, configuring nginx in Docker appears to be a more tedious task in general. On the other hand, if you have multiple hardware setups and need to synchronize deployments easily, using dockerized nginx might be a good option.

There is an excellent article which covers this topic in more detail and describes (dis-)advantages of both solutions [20].

2.5.3 Docker cache invalidation and command order

You could also familiarize yourself with how Docker image building works.

In layman's terms, when writing a Dockerfile, the more files change (e.g. actual source code), the further down the Dockerfile should the instructions for handling those files be. OS libraries go first, project dependencies afterwards, source code next, and finally, the application launching code.

This is because Docker caches every layer (a.k.a. a single Dockerfile instruction) internally based on the state of the files inside image root, and if some file is changed, the image rebuilding process starts over from the first instruction which handles that file, since the libraries installed before in Dockerfile remain unchanged.

Every layer depends on previous layer, so if cache for one layer is invalidated (e.g. changed files), the rest of the layers are too, and all following instructions are run anew (including less related stuff such as OS manipulation and libraries install).

Read up more on Docker cache here: [21].

2.5.4 Parent and child image caveats in Docker

2.5.5 Docker networks

3 Domain Model

3.1 Creating entities

[19] INSERT IMAGE LINK
[20] INSERT ARTICLE HERE
[21] INSERT DOCKER HELP LINK