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

Daniil Kraynov – parkanaur.net

February 16, 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 might 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 structure

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

[1] https://fastapi.tiangolo.com

   (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. [semi-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.

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!

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

This also installs **uvicorn[standard]** as per FastAPI installation tutorial.

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

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

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.:

3 Domain Model and Data Source

3.1 Creating entities