Getting Started

Part 1, Chapter 3

« Changelog | Docker Config »

In this chapter, we'll set up the base project structure.

Setup

Create a new project and install FastAPI along with Uvicorn, an ASGI server used to serve up FastAPI:

```
$ mkdir fastapi-tdd-docker && cd fastapi-tdd-docker
$ mkdir project && cd project
$ mkdir app
$ python3.9 -m venv env
$ source env/bin/activate
(env)$ pip install fastapi==0.62.0
(env)$ pip install uvicorn==0.13.1
```

> Feel free to swap out virtualenv and Pip for Poetry or Pipenv. For more, review Modern Python Environments.

Add an *__init__.py* file to the "app" directory along with a *main.py* file. Within *main.py*, create a new instance of FastAPI and set up a synchronous sanity check route:

```
# project/app/main.py


from fastapi import FastAPI

app = FastAPI()


@app.get("/ping")
def pong():
    return {"ping": "pong!"}
```

That's all you need to get a basic route up and running!

You should now have:

```
└── project
    └── app
        ├── __init__.py
        └── main.py
```

Run the server from the "project" directory:

```
(env)$ uvicorn app.main:app

INFO:     Started server process [84172]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
```

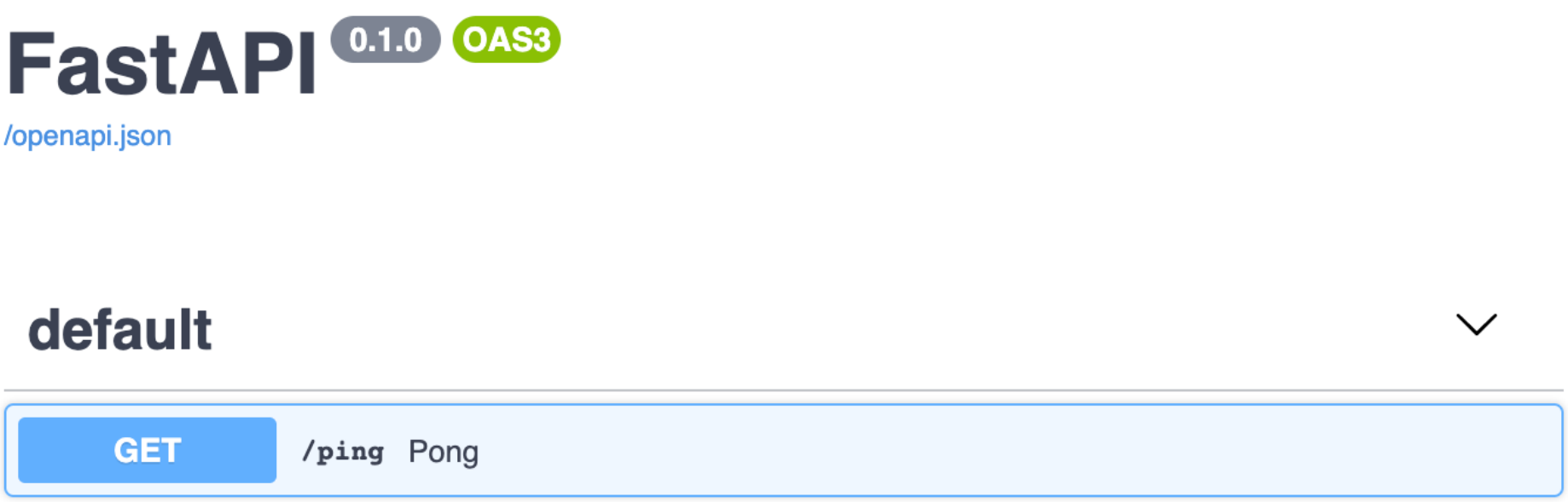
`app.main:app` tells Uvicorn where it can find the FastAPI application -- e.g., "within the 'app' module, you'll find the app, `app = FastAPI()`, in the 'main.py' file.

Navigate to http://localhost:8000/ping in your browser. You should see:

```
{
  "ping": "pong!"
}
```

> Why did we use Uvicorn to serve up FastAPI rather than a development server?
>
> Unlike Django or Flask, FastAPI does not have a built-in development server. This is both a positive and a negative in my opinion. On the one hand, it does take a bit more to serve up the app in development mode. On the other, this helps to conceptually separate the web framework from the web server, which is often a source of confusion for beginners when one moves from development to production with a web framework that does have a built-in development server.
>
> New to ASGI? Read through the excellent Introduction to ASGI: Emergence of an Async Python Web Ecosystem blog post.

FastAPI automatically generates a schema based on the OpenAPI standard. You can view the raw JSON at http://localhost:8000/openapi.json. This can be used to automatically generate client-side code for a front-end or mobile application. FastAPI uses it along with Swagger UI to create interactive API documentation, which can be viewed at http://localhost:8000/docs:



Kill the server.

Auto-reload

Let's run the app again. This time, we'll enable auto-reload mode so that the server will restart after changes are made to the code base:

```
(env)$ uvicorn app.main:app --reload

INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO:     Started reloader process [84187]
INFO:     Started server process [84189]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
```

Now when you make changes to the code, the app will automatically reload. Try this out.

Config

Add a new file called *config.py* to the "app" directory, where we'll define environment-specific configuration variables:

```python
# project/app/config.py


import logging
import os

from pydantic import BaseSettings


log = logging.getLogger("uvicorn")


class Settings(BaseSettings):
    environment: str = os.getenv("ENVIRONMENT", "dev")
    testing: bool = os.getenv("TESTING", 0)


def get_settings() -> BaseSettings:
    log.info("Loading config settings from the environment...")
    return Settings()
```

Here, we defined a `Settings` class with two attributes:

1. `environment` - defines the environment (i.e., dev, stage, prod)
2. `testing` - defines whether or not we're in test mode

BaseSettings, from Pydantic, validates the data so that when we create an instance of `Settings`, `environment` and `testing` will have types of `str` and `bool`, respectively.

Update *main.py* like so:

```python
# project/app/main.py


from fastapi import FastAPI, Depends

from app.config import get_settings, Settings


app = FastAPI()


@app.get("/ping")
def pong(settings: Settings = Depends(get_settings)):
    return {
        "ping": "pong!",
        "environment": settings.environment,
        "testing": settings.testing
    }
```

Take note of `settings: Settings = Depends(get_settings)`. Here, the `Depends` function is a dependency that declares another dependency, `get_settings`. Put another way, `Depends` depends on the result of `get_settings`. The value returned, `Settings`, is then assigned to the `settings` parameter.

If you're new to dependency injection, review the Dependencies guide from the offical FastAPI docs.

Run the server again. Navigate to http://localhost:8000/ping again. This time you should see:

```
{
  "ping": "pong!",
  "environment": "dev",
  "testing": false
}
```

Kill the server and set the following environment variables:

```
(env)$ export ENVIRONMENT=prod
(env)$ export TESTING=1
```

Run the server. Now, at http://localhost:8000/ping, you should see:

Feedback

```
{
  "ping": "pong!",
  "environment": "prod",
  "testing": true
}
```

What happens when you set the `TESTING` environment variable to `foo`? Try this out. Then update the variable to `0`.

With the server running, navigate to http://localhost:8000/ping and then refresh a few times. Back in your terminal, you should see several log messages for:

```
Loading config settings from the environment...
```

Essentially, `get_settings` gets called for each request. If we refactored the config so that the settings were read from a file, instead of from environment variables, it would be much too slow.

Let's use lru_cache to cache the settings so `get_settings` is only called once.

Update *config.py*:

```
# project/app/config.py

import logging
import os
from functools import lru_cache

from pydantic import BaseSettings


log = logging.getLogger("uvicorn")


class Settings(BaseSettings):
    environment: str = os.getenv("ENVIRONMENT", "dev")
    testing: bool = os.getenv("TESTING", 0)


@lru_cache()
def get_settings() -> BaseSettings:
    log.info("Loading config settings from the environment...")
    return Settings()
```

After the auto-reload, refresh the browser a few times. You should only see one `Loading config settings from the environment...` log message.

Async Handlers

Let's convert the synchronous handler over to an asynchronous one.

Rather than having to go through the trouble of spinning up a task queue (like Celery or RQ) or utilizing threads, FastAPI makes it easy to deliver routes asynchronously. As long as you don't have any blocking I/O calls in the handler, you can simply declare the handler as asynchronous by adding the `async` keyword like so:

```
@app.get("/ping")
async def pong(settings: Settings = Depends(get_settings)):
    return {
        "ping": "pong!",
        "environment": settings.environment,
        "testing": settings.testing
    }
```

That's it. Update the handler in your code, and then make sure it still works as expected.

Kill the server once done. Exit then remove the virtual environment as well. Then, add a *requirements.txt* file to the "project" directory:

```
fastapi==0.62.0
uvicorn==0.13.1
```

Finally, add a *.gitignore* to the project root:

```
__pycache__
env
```

You should now have:

```
├── .gitignore
└── project
    ├── app
    |   ├── __init__.py
    |   ├── config.py
    |   └── main.py
    └── requirements.txt
```

Init a git repo and commit your code.

« Changelog | Docker Config »

✓ Mark as Completed

TestDriven.io is a proud supporter of open source.

10% of profits from our FastAPI and Flask Web Development courses will be donated to the FastAPI and Flask teams, respectively.
Follow our contributions.

Developed by Michael Herman.

Follow @testdrivenio

Feedback