FastAPI

Table of Contents

Introduction

FastAPI is a modern API framework that boasts exceptionally high performance. At the time of writing, the project is less than two years old, yet it gained massive popularity in a relatively short time.

As the name suggests, FastAPI is excellent for building web APIs that typically returns JSON for application to application exchange. However, it also lends itself well for web applications, which generate HTML for browsers and user interaction.

Because FastAPI is modern, and it ships with modern features by default, such as "async"/ "await" and ASGI servers for building more scalable applications. FastAPI is an exciting project, gaining popularity. Let us jump in with a quick introduction on how to get started with FastAPI.

It would be not polite to break tradition and not do a Hello World example. Actual projects will be organised logically into isolated parts, which is covered later.

Before getting into it, I would like to credit Michael Kennedy, who hosts the awesome Talk Python To Me podcast and principle author of Talk Python Training. I owe my gratitude to Michael, whom I have great admiration for, and his content provided the majority of my understanding of FastAPI.

Environment

I use Red Hat Enterprise Linux 8 as my daily driver for stability, you can enable and use a more current version of Python 3 by enabling a module stream:

```
$ sudo dnf module enable python38
```

Create a new project working directory:

```
$ mkdir exampleforyou && cd exampleforyou
```

Create a new virtual environment, explicitly referencing, in this case Python 3.8:

```
$ python3.8 -m venv venv
```

Activate the virtual environment and install `fastapi` and `unicorn`. Uvicorn is a lightning-fast ASGI server implementation used to run the application:

```
$ source venv/bin/activate

$ pip install --upgrade pip
$ pip install fastapi uvicorn
```

Hello FastAPI

Create a `main.py`, this is the most basic example:

```
$ vi main.py
```

```
import fastapi
import uvicorn

motd = fastapi.FastAPI()


@motd.get('/')
def message():
    return {
        'message': "Hello FastAPI!"
    }


if __name__ == '__main__':
    uvicorn.run(motd, host='127.0.0.1', port=8000)
```

The example includes the two imports for FastAPI and a Unicorn, a server to run the application. Then it creates an instance of a FastAPI object called hello and defines a function decorated with the HTTP verb GET. Finally, it starts using the FastAPI object and optionally the host and port definition.

The application can be started either using `uvicorn`:

```
$ uvicorn main:motd --reload --port 8000
```

Or executing the `main.py`:

```
$ python main.py
```

Add Routing

It is a good idea to structure a project from the beginning. APIs can live under a directory and configured using a router.

Create a directory in the project:

```
$ mkdir api
```

Move and modify the decorator in this case also adding a new context path:

```
$ vi api/motd.py
```

```
import fastapi

router = fastapi.APIRouter()


@router.get('/api/motd')
def message():
    return {
        'message': "Hello FastAPI!"
    }
```

And update `main.py`, for clarity calling the FastAPI object `main_app` and some restructure using the function `configure()` to include APIs:

```
$ vi main.py
```

```
import fastapi
import uvicorn

from api import motd

main_app = fastapi.FastAPI()


def configure():
    configure_routing()


def configure_routing():
    main_app.include_router(motd.router)


if __name__ == '__main__':
    configure()
    uvicorn.run(main_app, host='127.0.0.1', port=8000)
else:
    configure()
```

This structure lays the foundation for a project to evolve and mature.

The application can be started either using uvicorn:

```
$ uvicorn main:main_app --reload --port 8000
```

Or executing the main.py:

```
$ python main.py
```

The home page will now display "Not Found" but the API is now available at http://127.0.0.1:8000/api/motd.

Add a home page

FastAPI support Jinja2 templates for rendering HTML

To use Jinja2 templates, install the package:

```
$ pip install jinja2
```

Create a `views` and `templates` directory in the project:

```
$ mkdir views templates
```

Using Jinja2 means you can break the HTML into reusable fragments, this should seem a familiar pattern from other web frameworks. Add a basic HTML template for base and home:

```
$ vi templates/_base.html
```

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>FastAPI</title>
</head>
<body>
    {% block content %}
    {% endblock %}
</body>
</html>
```

```
$ vi templates/home.html
```

```
{% extends "_base.html" %}

{% block content %}

<h1>Hello FastAPI!</h1>
<a href="/api/motd">Message Of The Day API</a>

{% endblock %}
```

Using template `TemplateResponse` add a home view `home.py`:

```
$ vi views/home.py
```

```
import fastapi
from starlette.requests import Request
from starlette.templating import Jinja2Templates

router = fastapi.APIRouter()
templates = Jinja2Templates('templates')


@router.get('/')
def home(request: Request):
    return templates.TemplateResponse('home.html', {'request': request})
```

Update `main.py` to import the home view and configure the routing:

```
$ vi main.py
```

```
import fastapi
import uvicorn
from api import motd
from views import home # New

main_app = fastapi.FastAPI()

def configure():
    configure_routing()


def configure_routing():
    main_app.include_router(motd.router)
    main_app.include_router(home.router) # New


if __name__ == '__main__':
    configure()
    uvicorn.run(main_app, host='127.0.0.1', port=7000)
else:
    configure()
```

Running the server and visiting http://127.0.0.1:8000 now should return a regular HTML page.

Static files

To include a static directory to include style sheets and images for example, FastAPI uses a mount concept.

Install the dependency for mounting `aiofiles`:

```
$ pip install aiofiles
```

Create a directory to keep static files, this example demonstrates using an image:

```
$ mkdir -p static/img
```

I copied an image called `fastapi_logo.png` into `static/img`

```
$ vi main.py
```

Add the following import:

```
from starlette.staticfiles import StaticFiles
```

Add the following `/static` mount in the `configure_routing()` function:

```
def configure_routing():
    main_app.mount('/static', StaticFiles(directory='static'), name='static') # New
    main_app.include_router(motd.router)
    main_app.include_router(home.router)
```

In the Jinga2 HTML template, static files can be referenced, in `templates/home.html` for example, like this:

```
<img src="static/img/fastapi_logo.png" alt="FastAPI">
```

Environment Variables

There are many approaches to managing secret variables such as passwords and access tokens. With the long term in mind, I prefer to using environment variables. This approach avoids ever including such secret in source code by mistake and set a good foundation for using containers at a later stage.

```
$ pip install environs
```

In this example set a local environment variable `ENV_SECRET`:

```
$ export ENV_SECRET="MyTopSecretToken"
```

Edit `main.py`:

```
$ vi main.py
```

Add the following import:

```
from environs import Env
```

And the following function:

```
def configure_env_vars():
    env = Env()
    env.read_env()
    if not env("ENV_SECRET"):
        print(f"WARNING: environment variable ENV_SECRET not found")
        raise Exception("environment variable ENV_SECRET not found.")
    else:
        home.secret = env("ENV_SECRET")
```

This new function needs calling so add it to the function `configure()`:

```
def configure():
    configure_routing()
    configure_env_vars()
```

If the environment variable `ENV_SECRET` is set, the function sets the value of `home.secret`, so add the following to `views/home.py`:

```
$ vi views/home.py
```

The following import:

```
from typing import Optional
```

And define the optional secret:

```
secret: Optional[str] = None
```

To test it works and to see how values can be passed to a template, update the view to return the value:

```
@router.get('/')
def home(request: Request):
    return templates.TemplateResponse('home.html', {'request': request,
'display_secret': secret})
```

Finally, edit the home template to display the secret:

```
$ vi templates/home.html
```

```
<p>SECRET: {{ display_secret }}</p>
```

Running the application now should display the secret on the home page. This demonstrates how values can be obtained in a safe way based on the environment and decouple environmental differences. When using Docker, Podman, Kubernetes or OpenShift this approach will pay dividends.

Environs also supports reading a hidden file called `.env` in the root of the project, rather that exporting variables at the user level, they can be defined at a project level. Just remember never to include `.env` in version control!

```
$ vi .env
```

```
export ENV_SECRET="MyTopSecretToken"
```

Digital Ocean API

Building upon everything demonstrated so far; this section will add a service that makes a call to an external API, using Digital Ocean to return a list of all the available droplet images. The API call to Digital Ocean requires authentication using an API token.

This approach uses another package dependency `httpx`:

```
$ pip install httpx
```

Make a new directory called services:

```
$ mkdir services
```

Add a new Python file for the Digital Ocean Service:

```
$ vi services/digital_ocean_service.py
```

```
from typing import Optional

import httpx

do_api_token: Optional[str] = None


async def get_droplet_images_async():
    url = f'https://api.digitalocean.com/v2/images?type=distribution'
    url_headers = {'Authorization': 'Bearer ' + do_api_token}

    async with httpx.AsyncClient() as client:
        resp = await client.get(url, headers=url_headers)

    data = resp.json()
    droplet_images = data['images']

    return droplet_images
```

Add a new API that consumes the service:

```
$ vi api/digital_ocean_images.py
```

```
import fastapi

from services import digital_ocean_service

router = fastapi.APIRouter()


@router.get('/api/droplet_images')
async def images():
    return await digital_ocean_service.get_droplet_images_async()
```

Update `main.py` to include the router of this new API and update the environment variable to set the Digital Ocean Access Token.

```
$ vi main.py
```

Include the imports:

```
from api import digital_ocean_images
from services import digital_ocean_service
```

Include the router:

```
def configure_routing():
    main_app.mount('/static', StaticFiles(directory='static'), name='static')
    main_app.include_router(motd.router)
    main_app.include_router(home.router)
    main_app.include_router(digital_ocean_images_api.router)
```

Update the function `configure_env_vars()`, you can keep adding `if not / else` blocks for numerous variables, for example:

```
def configure_env_vars():
    env = Env()
    env.read_env()
    if not env("ENV_SECRET"):
        print(f"WARNING: environment variable ENV_SECRET not found")
        raise Exception("environment variable ENV_SECRET not found.")
    else:
        home.secret = env("ENV_SECRET")
    if not env("DO_API_ACCESS_TOKEN"):
        print(f"WARNING: environment variable DO_API_ACCESS_TOKEN not found")
        raise Exception("environment variable DO_API_ACCESS_TOKEN not found.")
    else:
        digital_ocean_service.do_api_token = env("DO_API_ACCESS_TOKEN")
```

Remember to export the token environment variable or add it to `.env`, for example:

```
export DO_API_ACCESS_TOKEN=xyzxyzxyzyz
```

Visit http://127.0.0.1:8000/api/droplet_images to see the all JSON results returned.

Great, finally update the home view to display the results, in this case a list of the slugs for all the available distribution images at Digital Ocean.

```
$ vi views/home.py
```

Import the service `digital_ocean_service`:

```
from services import digital_ocean_service
```

And update the function, notice the function is converted using `async` and `await`:

```
@router.get('/')
async def home(request: Request):
    droplet_images = await digital_ocean_service.get_droplet_images_async()
    return templates.TemplateResponse('home.html', {'request': request,
'display_secret': secret, 'droplet_images': droplet_images})
```

Finally, update the home template to display the slugs:

```
$ vi templates/home.html
```

```
    {% for i in images %}
     <li> {{ i.slug }}</li>
    {% endfor %}
```

I think this has been a great introduction to FastAPI, covering the bases to wet ones appetite. The FastAPI documentation is great and the power and I believe the speed and simplicity of this web framework is a serious contender.