FastAPI

FastAPI is a modern, fast (high-performance), web framework for building APIs with Python 3.6+ based on standard Python type hints.

The key features are:

- Fast: Very high performance, on par with NodeJS and Go (thanks to Starlette and Pydantic). One of the fastest Python frameworks available.
- Fast to code: Increase the speed to develop features by about 200% to 300%.
- Fewer bugs: Reduce about 40% of human (developer) induced errors.
- Intuitive: Great editor support. Completion everywhere. Less time debugging.
- Easy: Designed to be easy to use and learn. Less time reading docs.
- Short: Minimize code duplication. Multiple features from each parameter declaration. Fewer bugs.
- Robust: Get production-ready code. With automatic interactive documentation.
- Standards-based: Based on (and fully compatible with) the open standards for APIs: OpenAPI (previously known as Swagger) and JSON Schema.
- Authentication with JWT: with a super nice tutorial on how to set it up.

Installation

```
pip install fastapi
```

You will also need an ASGI server, for production such as Uvicorn or Hypercorn.

```
pip install uvicorn[standard]
```

Simple example

- Create a file `main.py` with:

```
from typing import Optional

from fastapi import FastAPI
```

```
app = FastAPI()


@app.get("/")
def read_root():
    return {"Hello": "World"}


@app.get("/items/{item_id}")
def read_item(item_id: int, q: Optional[str] = None):
    return {"item_id": item_id, "q": q}
```

- Run the server:

```
uvicorn main:app --reload
```

- Open your browser at http://127.0.0.1:8000/items/5?q=somequery. You will see the JSON response as:

```
{
  "item_id": 5,
  "q": "somequery"
}
```

You already created an API that:

- Receives HTTP requests in the paths `/` and `/items/{item_id}`.
- Both paths take GET operations (also known as HTTP methods).
- The path `/items/{item_id}` has a path parameter `item_id` that should be an `int`.
- The path `/items/{item_id}` has an optional `str` query parameter `q`.
- Has interactive API docs made for you:
- Swagger: http://127.0.0.1:8000/docs.
- Redoc: http://127.0.0.1:8000/redoc.

You will see the automatic interactive API documentation (provided by Swagger UI):

Sending data to the server

When you need to send data from a client (let's say, a browser) to your API, you have three basic options:

- As path parameters in the URL (`/items/2`).
- As query parameters in the URL (`/items/2?skip=true`).

- In the body 🕮 of a POST request.

To send simple data use the first two, to send complex or sensitive data, use the last.

It also supports sending data through cookies and headers.

Path Parameters

You can declare path "parameters" or "variables" with the same syntax used by Python format strings:

```
@app.get("/items/{item_id}")
def read_item(item_id: int):
    return {"item_id": item_id}
```

If you define the type hints of the function arguments, FastAPI will use pydantic 🕮 data validation.

If you need to use a Linux path as an argument, check this workaround, but be aware that it's not supported by OpenAPI.

Order matters

Because path operations are evaluated in order, you need to make sure that the path for the fixed endpoint `/users/me` is declared before the variable one `/users/{user_id}`:

```
@app.get("/users/me")
async def read_user_me():
    return {"user_id": "the current user"}


@app.get("/users/{user_id}")
async def read_user(user_id: str):
    return {"user_id": user_id}
```

Otherwise, the path for `/users/{user_id}` would match also for `/users/me`, "thinking" that it's receiving a parameter user_id with a value of "me".

Predefined values

If you want the possible valid path parameter values to be predefined, you can use a standard Python `Enum`.

```
from enum import Enum


class ModelName(str, Enum):
    alexnet = "alexnet"
    resnet = "resnet"
    lenet = "lenet"


@app.get("/models/{model_name}")
def get_model(model_name: ModelName):
    if model_name == ModelName.alexnet:
        return {"model_name": model_name, "message": "Deep Learning FTW!"}

    if model_name.value == "lenet":
        return {"model_name": model_name, "message": "LeCNN all the images"}

    return {"model_name": model_name, "message": "Have some residuals"}
```

These are the basics, FastAPI supports more complex path parameters and string validations.

Query Parameters

When you declare other function parameters that are not part of the path parameters, they are automatically interpreted as "query" parameters.

```
fake_items_db = [{"item_name": "Foo"}, {"item_name": "Bar"}, {"item_name":
"Baz"}]


@app.get("/items/")
async def read_item(skip: int = 0, limit: int = 10):
    return fake_items_db[skip : skip + limit]
```

The query is the set of key-value pairs that go after the `?` in a URL, separated by `&` characters.

For example, in the URL: http://127.0.0.1:8000/items/?skip=0&limit=10

These are the basics, FastAPI supports more complex query parameters and string validations.

Request Body

To declare a request body, you use Pydantic models with all their power and benefits.

```
from typing import Optional
from pydantic import BaseModel


class Item(BaseModel):
    name: str
    description: Optional[str] = None
    price: float
    tax: Optional[float] = None


@app.post("/items/")
```

```
async def create_item(item: Item):
    return item
```

With just that Python type declaration, FastAPI will:

- Read the body of the request as JSON.
- Convert the corresponding types (if needed).
- Validate the data: If the data is invalid, it will return a nice and clear error, indicating exactly where and what was the incorrect data.
- Give you the received data in the parameter `item`.
- Generate JSON Schema definitions for your model.
- Those schemas will be part of the generated OpenAPI schema, and used by the automatic documentation UIs.

These are the basics, FastAPI supports more complex patterns such as:

- Using multiple models in the same query.
- Additional validations of the pydantic models.
- Nested models.

Sending data to the client

When you create a FastAPI path operation you can normally return any data from it: a `dict`, a `list`, a Pydantic model, a database model, etc.

By default, FastAPI would automatically convert that return value to JSON using the `jsonable_encoder`.

To return custom responses such as a direct string, xml or html use `Response`:

```
from fastapi import FastAPI, Response

app = FastAPI()


@app.get("/legacy/")
def get_legacy_data():
    data = """<?xml version="1.0"?>
    <shampoo>
    <Header>
        Apply shampoo here.
    </Header>
    <Body>
        You'll have to use soap here.
    </Body>
    </shampoo>
    """
    return Response(content=data, media_type="application/xml")
```

Handling errors

There are many situations in where you need to notify an error to a client that is using your API.

In these cases, you would normally return an HTTP status code in the range of 400 (from 400 to 499).

This is similar to the 200 HTTP status codes (from 200 to 299). Those "200" status codes mean that somehow there was a "success" in the request.

To return HTTP responses with errors to the client you use `HTTPException`.

```
from fastapi import HTTPException

items = {"foo": "The Foo Wrestlers"}


@app.get("/items/{item_id}")
async def read_item(item_id: str):
    if item_id not in items:
        raise HTTPException(status_code=404, detail="Item not found")
    return {"item": items[item_id]}
```

Updating data

Update replacing with PUT

To update an item you can use the HTTP PUT operation.

You can use the `jsonable_encoder` to convert the input data to data that can be stored as JSON (e.g. with a NoSQL database). For example, converting datetime to str.

```
from typing import List, Optional

from fastapi.encoders import jsonable_encoder
from pydantic import BaseModel


class Item(BaseModel):
    name: Optional[str] = None
    description: Optional[str] = None
    price: Optional[float] = None
    tax: float = 10.5
    tags: List[str] = []
```

```
items = {
    "foo": {"name": "Foo", "price": 50.2},
    "bar": {"name": "Bar", "description": "The bartenders", "price": 62,
"tax": 20.2},
    "baz": {"name": "Baz", "description": None, "price": 50.2, "tax": 10.5,
"tags": []},
}


@app.get("/items/{item_id}", response_model=Item)
async def read_item(item_id: str):
    return items[item_id]


@app.put("/items/{item_id}", response_model=Item)
async def update_item(item_id: str, item: Item):
    update_item_encoded = jsonable_encoder(item)
    items[item_id] = update_item_encoded
    return update_item_encoded
```

Partial updates with PATCH

You can also use the HTTP PATCH operation to partially update data.

This means that you can send only the data that you want to update, leaving the rest intact.

Configuration

Application configuration

In many cases your application could need some external settings or configurations, for example secret keys, database credentials, credentials for email services, etc.

You can load these configurations through environmental variables, or you can use the awesome Pydantic settings management, whose advantages are:

- Do Pydantic's type validation on the fields.
- Automatically reads the missing values from environmental variables.
- Supports reading variables from Dotenv files.
- Supports secrets.

First you define the `Settings` class with all the fields:

File: `config.py`:

```
from pydantic import BaseSettings
```

```
class Settings(BaseSettings):
    verbose: bool = True
    database_url: str = "tinydb://~/.local/share/pyscrobbler/database.tinydb"
```

Then in the api definition, set the dependency.

File: `api.py`:

```
from functools import lru_cache
from fastapi import Depends, FastAPI


app = FastAPI()


@lru_cache()
def get_settings() -> Settings:
    """Configure the program settings."""
    return Settings()


@app.get("/verbose")
def verbose(settings: Settings = Depends(get_settings)) -> bool:
    return settings.verbose
```

Where:

- `get_settings` is the dependency function that configures the `Settings` object. The endpoint `verbose` is dependant of `get_settings`.
- The `@lru_cache` decorator changes the function it decorates to return the same value that was returned the first time, instead of computing it again, executing the code of the function every time.

So, the function will be executed once for each combination of arguments. And then the values returned by each of those combinations of arguments will be used again and again whenever the function is called with exactly the same combination of arguments.

Creating the `Settings` object is a costly operation as it needs to check the environment variables or read a file, so we want to do it just once, not on each request.

This setup makes it easy to inject testing configuration so as not to break production code.

OpenAPI configuration

Define title, description and version

```
from fastapi import FastAPI
```

```
app = FastAPI(
    title="My Super Project",
    description="This is a very fancy project, with auto docs for the API and
everything",
    version="2.5.0",
)
```

Define path tags

You can add tags to your path operation, pass the parameter tags with a list of `str` (commonly just one `str`):

```
from typing import Optional, Set

from pydantic import BaseModel


class Item(BaseModel):
    name: str
    description: Optional[str] = None
    price: float
    tax: Optional[float] = None
    tags: Set[str] = []


@app.post("/items/", response_model=Item, tags=["items"])
async def create_item(item: Item):
    return item


@app.get("/items/", tags=["items"])
async def read_items():
    return [{"name": "Foo", "price": 42}]


@app.get("/users/", tags=["users"])
async def read_users():
    return [{"username": "johndoe"}]
```

They will be added to the OpenAPI schema and used by the automatic documentation interfaces.

ADD METADATA TO THE TAGS

```
tags_metadata = [
    {
        "name": "users",
        "description": "Operations with users. The **login** logic is also
here.",
    },
    {
        "name": "items",
        "description": "Manage items. So _fancy_ they have their own docs.",
        "externalDocs": {
            "description": "Items external docs",
            "url": "https://fastapi.tiangolo.com/",
        },
    },
]
```

app = FastAPI(openapi_tags=tags_metadata)

Add a summary and description

```
@app.post("/items/", response_model=Item, summary="Create an item")
async def create_item(item: Item):
    """
    Create an item with all the information:

    - **name**: each item must have a name
    - **description**: a long description
    - **price**: required
    - **tax**: if the item doesn't have tax, you can omit this
    - **tags**: a set of unique tag strings for this item
    """
    return item
```

Response description

```
@app.post(
    "/items/",
    response_description="The created item",
)
async def create_item(item: Item):
    return item
```

Deprecate a path operation

When you need to mark a path operation as deprecated, but without removing it

```
@app.get("/elements/", tags=["items"], deprecated=True)
async def read_elements():
    return [{"item_id": "Foo"}]
```

Deploy with Docker.

FastAPI has it's own optimized docker, which makes the deployment of your applications really easy.

- In your project directory create the `Dockerfile` file:

```
FROM tiangolo/uvicorn-gunicorn-fastapi:python3.7
```

```
COPY ./app /app
```

- Go to the project directory (in where your Dockerfile is, containing your app directory).
- Build your FastAPI image:

```
docker build -t myimage .
```

- Run a container based on your image:

```
docker run -d --name mycontainer -p 80:80 myimage
```

Now you have an optimized FastAPI server in a Docker container. Auto-tuned for your current server (and number of CPU cores).

Installing dependencies

If your program needs other dependencies, use the next dockerfile:

```
FROM tiangolo/uvicorn-gunicorn-fastapi:python3.7

COPY ./requirements.txt /app
RUN pip install -r requirements.txt

COPY ./app /app
```

Other project structures

The previous examples assume that you have followed the FastAPI project structure. If instead you've used mine your application will be defined in the `app` variable in the `src/program_name/entrypoints/api.py` file.

To make things simpler make the `app` variable available on the root of your package, so you can do `from program_name import app` instead of `from program_name.entrypoints.api import app`. To do that we need to add `app` to the `__all__` internal python variable of the `__init__.py` file of our package.

File: `src/program_name/__init__.py`:

```
from .entrypoints.ap
import app

__all__: List[str] = ['app']
```

The image is configured through environmental variables

So we will need to use:

```
FROM tiangolo/uvicorn-gunicorn-fastapi:python3.7

ENV MODULE_NAME="program_name"

COPY ./src/program_name /app/program_name
```

Testing

FastAPI gives a `TestClient` object borrowed from Starlette to do the integration tests on your application.

```
from fastapi import FastAPI
from fastapi.testclient import TestClient

app = FastAPI()


@app.get("/")
async def read_main():
    return {"msg": "Hello World"}


@pytest.fixture(name="client")
def client_() -> TestClient:
    """Configure FastAPI TestClient."""
    return TestClient(app)


def test_read_main(client: TestClient):
    response = client.get("/")
    assert response.status_code == 200
    assert response.json() == {"msg": "Hello World"}
```

Test a POST request

```
result = client.post(
    "/items/",
    headers={"X-Token": "coneofsilence"},
    json={"id": "foobar", "title": "Foo Bar", "description": "The Foo
Barters"},
)
```

Inject testing configuration

If your application follows the application configuration section 🕮, injecting testing configuration is easy with dependency injection.

Imagine you have a `db_tinydb` fixture that sets up the testing database:

```
@pytest.fixture(name="db_tinydb")
def db_tinydb_(tmp_path: Path) -> str:
    """Create an TinyDB database engine.

    Returns:
        database_url: Url used to connect to the database.
    """
    tinydb_file_path = str(tmp_path / "tinydb.db")
    return f"tinydb:///{tinydb_file_path}"
```

You can override the default `database_url` with:

```
@pytest.fixture(name="client")
def client_(db_tinydb: str) -> TestClient:
    """Configure FastAPI TestClient."""

    def override_settings() -> Settings:
        """Inject the testing database in the application settings."""
        return Settings(database_url=db_tinydb)

    app.dependency_overrides[get_settings] = override_settings
    return TestClient(app)
```

Add endpoints only on testing environment

Sometimes you want to have some API endpoints to populate the database for end to end testing the frontend. If your `app` config has the `environment` attribute, you could try to do:

```
app = FastAPI()


@lru_cache()
def get_config() -> Config:
    """Configure the program settings."""
    # no cover: the dependency are injected in the tests
    log.info("Loading the config")
    return Config()  # pragma: no cover


if get_config().environment == "testing":

    @app.get("/seed", status_code=201)
    def seed_data(
        repo: Repository = Depends(get_repo),
        empty: bool = True,
        num_articles: int = 3,
        num_sources: int = 2,
    ) -> None:
        """Add seed data for the end to end tests.

        Args:
            repo: Repository to store the data.
        """
        services.seed(
            repo=repo, empty=empty, num_articles=num_articles,
num_sources=num_sources
        )
        repo.close()
```

But the injection of the dependencies is only done inside the functions, so `get_config().environment` will always be the default value. I ended up doing that check inside the endpoint, which is not ideal.

```
@app.get("/seed", status_code=201)
def seed_data(
    config: Config = Depends(get_config),
    repo: Repository = Depends(get_repo),
    empty: bool = True,
    num_articles: int = 3,
    num_sources: int = 2,
) -> None:
    """Add seed data for the end to end tests.

    Args:
        repo: Repository to store the data.
    """
    if config.environment != "testing":
        repo.close()
        raise HTTPException(status_code=404)
    ...
```

Tips and tricks

Create redirections

Returns an HTTP redirect. Uses a 307 status code (Temporary Redirect) by default.

```
from fastapi import FastAPI
from fastapi.responses import RedirectResponse

app = FastAPI()


@app.get("/typer")
async def read_typer():
    return RedirectResponse("https://typer.tiangolo.com")
```

Test that your application works locally

Once you have your application built and tested, everything should work right? well, sometimes it don't. If you need to use `pdb` to debug what's going on, you can't use the docker as you won't be able to interact with the debugger.

Instead, launch an uvicorn application directly with:

```
uvicorn program_name:app --reload
```

Note: The command is assuming that your `app` is available at the root of your package, look at the deploy section if you feel lost.

Resolve the 307 error

Probably you've introduced an ending `/` to the endpoint, so instead of asking for `/my/endpoint` you tried to do `/my/endpoint/`.

Resolve the 409 error

Probably an exception was raised in the backend, use `pdb` to follow the trace and catch where it happened.

Resolve the 422 error

You're probably passing the wrong arguments to the POST request, to solve it see the `text` attribute of the result. For example:

```
# client: TestClient
result = client.post(
    "/source/add",
    json={"body": body},
)

result.text
# '{"detail":[{"loc":["query","url"],"msg":"field
required","type":"value_error.missing"}]}'
```

The error is telling us that the required `url` parameter is missing.

Logging

By default the application log messages are not shown in the uvicorn log, you need to add the next lines to the file where your app is defined:

File: `src/program_name/entrypoints/api.py`:

```
from fastapi import FastAPI
from fastapi.logger import logger
import logging

log = logging.getLogger("gunicorn.error")
logger.handlers = log.handlers
if __name__ != "main":
    logger.setLevel(log.level)
else:
    logger.setLevel(logging.DEBUG)

app = FastAPI()

# rest of the application...
```

Logging to Sentry

FastAPI can integrate with Sentry or similar application loggers through the ASGI middleware.

Run a FastAPI server in the background for testing purposes

Sometimes you want to launch a web server with a simple API to test a program that can't use the testing client. First define the API to launch with:

File: `tests/api_server.py`:

```
from fastapi import FastAPI, HTTPException

app = FastAPI()


@app.get("/existent")
async def existent():
    return {"msg": "exists!"}


@app.get("/inexistent")
async def inexistent():
    raise HTTPException(status_code=404, detail="It doesn't exist")
```

Then create the fixture:

File: `tests/conftest.py`:

```
from multiprocessing import Process
from typing import Generator
import pytest
import uvicorn

from .api_server import app


def run_server() -> None:
    """Command to run the fake api server."""
    uvicorn.run(app)


@pytest.fixture()
def _server() -> Generator[None, None, None]:
    """Start the fake api server."""
    proc = Process(target=run_server, args=(), daemon=True)
    proc.start()
    yield
    proc.kill()  # Cleanup after test
```

Now you can use the `server: None` fixture in your tests and run your queries against `http://localhost:8000`.

Interesting features to explore

- Structure big applications.
- Dependency injection.
- Running background tasks after the request is finished.
- Return a different response model.
- Upload files.
- Set authentication.
- Host behind a proxy.
- Static files.

Issues

- FastAPI does not log messages: update `pyscrobbler` and any other maintained applications and remove the snippet defined in the logging section.

References

- Docs

- Git
- Awesome FastAPI
- Testdriven.io course: suggested by the developer.

Last update: 2022-11-24