

Fast API 最佳实践

<https://github.com/zhanymkanov/fastapi-best-practices>

1. Project Structure. Consistent & predictable

- If looking at the project structure doesn't give you an idea of what the project is about, then the structure might be unclear.
- If you have to open packages to understand what modules are located in them, then your structure is unclear.
- If the frequency and location of the files feels random, then your project structure is bad.
- If looking at the module's location and its name doesn't give you an idea of what's inside it, then your structure is very bad.

一个项目示例：

```
fastapi-project
├── alembic/
├── src
│   ├── auth
│   │   ├── router.py
│   │   ├── schemas.py # pydantic models
│   │   ├── models.py # db models
│   │   ├── dependencies.py
│   │   ├── config.py # local configs
│   │   ├── constants.py
│   │   ├── exceptions.py
│   │   ├── service.py
│   │   └── utils.py
│   ├── aws
│   │   └── client.py # client model for external service
communication
│   ├── schemas.py
│   ├── config.py
│   ├── constants.py
│   ├── exceptions.py
│   └── utils.py
└── posts
    ├── router.py
    ├── schemas.py
    ├── models.py
    ├── dependencies.py
    ├── constants.py
    ├── exceptions.py
    ├── service.py
    └── utils.py
├── config.py # global configs
├── models.py # global models
├── exceptions.py # global exceptions
├── pagination.py # global module e.g. pagination
├── database.py # db connection related stuff
└── main.py
├── tests/
│   ├── auth
│   ├── aws
│   └── posts
├── templates/
│   └── index.html
└── requirements
```

1. Store all domain directories inside `src` foldersrc/ - highest level of an app, contains common models, configs, and constants, etc.
 - a. `src/main.py` - root of the project, which inits the FastAPI app
2. Each package has its own router, schemas, models, etc.[router.py](#) - is a core of each module with all the endpoints
 - a. [schemas.py](#) - for pydantic models
 - b. [models.py](#) - for db models
 - c. [service.py](#) - module specific business logic
 - d. [dependencies.py](#) - router dependencies
 - e. [constants.py](#) - module specific constants and error codes
 - f. [config.py](#) - e.g. env vars
 - g. [utils.py](#) - non-business logic functions, e.g. response normalization, data enrichment, etc.
 - h. [exceptions.py](#) - module specific exceptions, e.g. `PostNotFound`, `InvalidUserData`
3. When package requires services or dependencies or constants from other packages - import them with an explicit module name

```
from src.auth import constants as auth_constants
from src.notifications import service as notification_service
from src.posts.constants import ErrorCode as PostsErrorCode # in
case we have Standard ErrorCode in constants module of each
package
```

2. 充分使用 pydantic 进行数据校验

Pydantic has a rich set of features to validate and transform data.

In addition to regular features like required & non-required fields with default values, Pydantic has built-in comprehensive data processing tools like regex, enums for limited allowed options, length validation, email validation, etc.

待扩展：pydantic 常用的校验特性

3. 使用 dependencies 校验 DB 数据

Pydantic 只能验证来自客户端输入的值。使用依赖项根据数据库约束（如电子邮件已存在、找不到用户等）验证数据。比如校验某个 ID 对应的记录是否存在就可以考虑这种思想，否则在每个接口逻辑里面都要去判断是否存在记录

```

# dependencies.py
async def valid_post_id(post_id: UUID4) -> Mapping:
    post = await service.get_by_id(post_id)
    if not post:
        raise PostNotFound()

    return post

# router.py
@router.get("/posts/{post_id}", response_model=PostResponse)
async def get_post_by_id(post: Mapping = Depends(valid_post_id)):
    return post

@router.put("/posts/{post_id}", response_model=PostResponse)
async def update_post(
    update_data: PostUpdate,
    post: Mapping = Depends(valid_post_id),
):
    updated_post: Mapping = await service.update(id=post["id"],
data=update_data)
    return updated_post

@router.get("/posts/{post_id}/reviews",
response_model=list[ReviewsResponse])
async def get_post_reviews(post: Mapping =
Depends(valid_post_id)):
    post_reviews: list[Mapping] = await
reviews_service.get_by_post_id(post["id"])
    return post_reviews

```

4. 多使用 dependencies 链

Dependencies can use other dependencies and avoid code repetition for similar logic.

重复利用的逻辑拆分出来作为依赖项，减少代码重复

5. 解耦、重复使用依赖性，依赖项的结果会把缓存

Dependencies can be reused multiple times, and they won't be recalculated - FastAPI caches dependency's result within a request's scope by default, i.e. if we have a dependency that calls service `get_post_by_id`, we won't be visiting DB each time we call this dependency - only the first function call.

Knowing this, we can easily decouple dependencies onto multiple smaller functions that operate on a smaller domain and are easier to reuse in other routes. For example, in the code below we are using `parse_jwt_data` three times:

1. `valid_owned_post`
2. `valid_active_creator`
3. `get_user_post,`

but `parse_jwt_data` is called only once, in the very first call.

```

# dependencies.py
from fastapi import BackgroundTasks
from fastapi.security import OAuth2PasswordBearer
from jose import JWTError, jwt

async def valid_post_id(post_id: UUID4) -> Mapping:
    post = await service.get_by_id(post_id)
    if not post:
        raise PostNotFound()

    return post

async def parse_jwt_data(
    token: str =
Depends(OAuth2PasswordBearer(tokenUrl="/auth/token"))
) -> dict:
    try:
        payload = jwt.decode(token, "JWT_SECRET",
algorithm="HS256")
    except JWTError:
        raise InvalidCredentials()

    return {"user_id": payload["id"]}

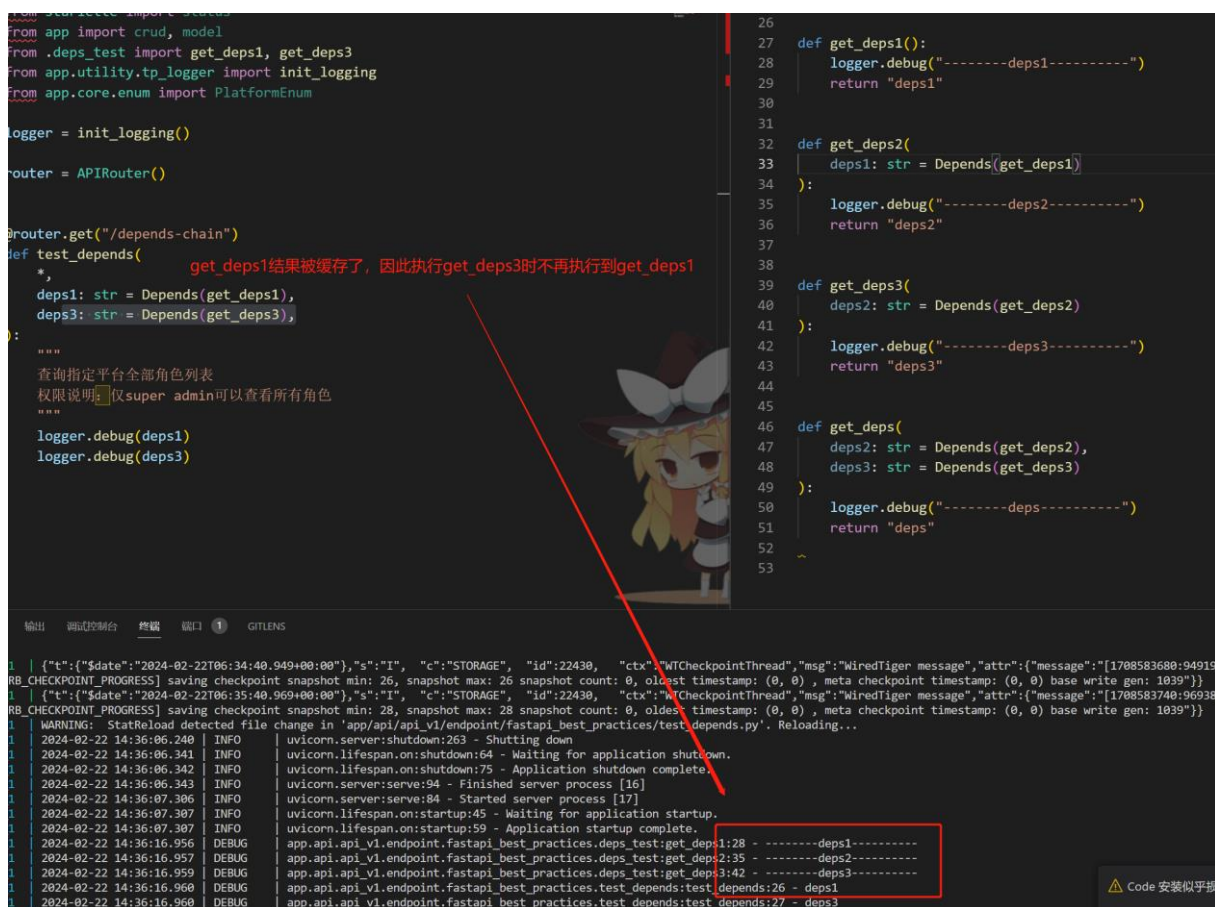
async def valid_owned_post(
    post: Mapping = Depends(valid_post_id),
    token_data: dict = Depends(parse_jwt_data),
) -> Mapping:
    if post["creator_id"] != token_data["user_id"]:
        raise UserNotOwner()

    return post

async def valid_active_creator(
    token_data: dict = Depends(parse_jwt_data),
):
    user = await users_service.get_by_id(token_data["user_id"])
    if not user["is_active"]:
        raise UserIsBanned()

```

本地试验如下：



```
from starlette import status
from app import crud, model
from .deps_test import get_deps1, get_deps3
from app.utility.tp_logger import init_logging
from app.core.enum import PlatformEnum

logger = init_logging()

router = APIRouter()

router.get("/depends-chain")
def test_depends(
    *,
    deps1: str = Depends(get_deps1),
    deps3: str = Depends(get_deps3),
):
    """
    查询指定平台全部角色列表
    权限说明: 仅super admin可以查看所有角色
    """
    logger.debug(deps1)
    logger.debug(deps3)

26
27 def get_deps1():
28     logger.debug("-----deps1-----")
29     return "deps1"
30
31
32 def get_deps2(
33     deps1: str = Depends(get_deps1)
34 ):
35     logger.debug("-----deps2-----")
36     return "deps2"
37
38
39 def get_deps3(
40     deps2: str = Depends(get_deps2)
41 ):
42     logger.debug("-----deps3-----")
43     return "deps3"
44
45
46 def get_deps(
47     deps2: str = Depends(get_deps2),
48     deps3: str = Depends(get_deps3)
49 ):
50     logger.debug("-----deps-----")
51     return "deps"
52
53

get_deps1结果被缓存了, 因此执行get_deps3时不再执行到get_deps1

WARNING: StatReload detected file change in 'app/api/api_v1/endpoint/fastapi_best_practices/test_depends.py'. Reloading...
2024-02-22 14:36:06.240 INFO | uvicorn.server:shutdown:263 - Shutting down
2024-02-22 14:36:06.341 INFO | uvicorn.lifespan.on:shutdown:64 - Waiting for application shutdown.
2024-02-22 14:36:06.342 INFO | uvicorn.lifespan.on:shutdown:75 - Application shutdown complete.
2024-02-22 14:36:06.343 INFO | uvicorn.server:serve:94 - Finished server process [16]
2024-02-22 14:36:07.386 INFO | uvicorn.server:serve:84 - Started server process [17]
2024-02-22 14:36:07.387 INFO | uvicorn.lifespan.on:startup:45 - Waiting for application startup.
2024-02-22 14:36:07.387 INFO | uvicorn.lifespan.on:startup:59 - Application startup complete.
2024-02-22 14:36:16.956 DEBUG | app.api.api_v1.endpoint.fastapi_best_practices.deps_test:get_deps1:28 - -----deps1-----
2024-02-22 14:36:16.957 DEBUG | app.api.api_v1.endpoint.fastapi_best_practices.deps_test:get_deps2:35 - -----deps2-----
2024-02-22 14:36:16.959 DEBUG | app.api.api_v1.endpoint.fastapi_best_practices.deps_test:get_deps3:42 - -----deps3-----
2024-02-22 14:36:16.960 DEBUG | app.api.api_v1.endpoint.fastapi_best_practices.test_depends:26 - deps1
2024-02-22 14:36:16.960 DEBUG | app.api.api_v1.endpoint.fastapi_best_practices.test_depends:27 - deps3
```

6. 遵从 REST 风格

7. Don't make your routes async, if you have only blocking

I/O operations 根据任务的类型（I/O 密集型还是 CPU 密集型）选择合适的并发模型（异步、多线程或多进程）

Under the hood, FastAPI can [effectively handle](#) both async and sync I/O operations.

- FastAPI runs sync routes in the [threadpool](#) and blocking I/O operations won't stop the [event loop](#) from executing the tasks.
- Otherwise, if the route is defined async then it's called regularly via await and FastAPI trusts you to do only non-blocking I/O operations.

The caveat is if you fail that trust and execute blocking operations within async routes, the

event loop will not be able to run the next tasks until that blocking operation is done.

FastAPI 是一个现代的、快速（高性能）的 Web 框架，用于构建 APIs。其基于 Python 3.6 类型提示，基于 Starlette 进行异步编程，FastAPI 运行在 ASGI 上。

FastAPI 可以处理同步和异步的路由处理函数。对于这两种类型的函数，FastAPI 的处理方式有所不同：

1. **同步路由：**当你定义一个同步路由（即没有使用 `async def` 定义的函数），FastAPI 会在一个线程池中运行这个函数。这意味着，如果你的函数中有阻塞的 I/O 操作（例如，读写文件或数据库查询），这个操作会阻塞当前的线程，但不会阻塞事件循环。事件循环可以继续处理其他任务。这是因为每个同步任务都在自己的线程中运行，所以一个任务的阻塞不会影响其他任务。

2. **异步路由：**当你定义一个异步路由（即使用 `async def` 定义的函数），FastAPI 会直接调用这个函数，并期望它执行非阻塞的 I/O 操作。因为在异步编程中，`await` 关键字会将控制权交还给事件循环，事件循环可以继续处理其他任务，直到异步操作完成。如果你在异步路由中执行阻塞的 I/O 操作，那么你就违反了约定，事件循环会被阻塞，无法处理其他任务，这会降低应用的性能。

总的来说，FastAPI 通过在线程池中运行同步路由，并期望异步路由只执行非阻塞的 I/O 操作，来确保应用的高性能和响应能力。

示例：

```

import asyncio
import time

@router.get("/terrible-ping")
async def terrible_catastrophic_ping():
    time.sleep(10) # I/O blocking operation for 10 seconds
    pong = service.get_pong() # I/O blocking operation to get
    pong from DB

    return {"pong": pong}

@router.get("/good-ping")
def good_ping():
    time.sleep(10) # I/O blocking operation for 10 seconds, but
    in another thread
    pong = service.get_pong() # I/O blocking operation to get
    pong from DB, but in another thread

    return {"pong": pong}

@router.get("/perfect-ping")
async def perfect_ping():
    await asyncio.sleep(10) # non-blocking I/O operation
    pong = await service.async_get_pong() # non-blocking I/O db
    call

    return {"pong": pong}

```

await 在 Python 的异步编程中是非阻塞的操作。**await** 关键字用于等待一个异步操作的结果。在等待的过程中，它不会阻塞整个程序的执行。

当你在 Python 程序中使用 **await** 时，程序会在当前的协程（coroutine）中暂停执行，然后切换到事件循环，继续执行其他的协程。当被 **await** 的操作完成时，程序会再次切换回原来的协程，继续执行后面的代码。

这就是为什么 **await** 被称为非阻塞的操作。尽管它会暂停当前的协程，但是它不会阻塞整个程序的

执行。相反，它允许程序在等待一个操作的结果时，继续执行其他的任务。

What happens when we call:

1. GET /terrible-pingFastAPI server receives a request and starts handling it
 - a. Server's event loop and all the tasks in the queue will be waiting until `time.sleep()` is finished Server thinks `time.sleep()` is not an I/O task, so it waits until it is finished
 - i. Server won't accept any new requests while waiting
 - b. Then, event loop and all the tasks in the queue will be waiting until `service.get_pong` is finishedServer thinks `service.get_pong()` is not an I/O task, so it waits until it is finished
 - i. Server won't accept any new requests while waiting
 - c. Server returns the response.After a response, server starts accepting new requests
2. GET /good-pingFastAPI server receives a request and starts handling it
 - a. FastAPI sends the whole route `good_ping` to the threadpool, where a worker thread will run the function
 - b. While `good_ping` is being executed, event loop selects next tasks from the queue and works on them (e.g. accept new request, call db)Independently of main thread (i.e. our FastAPI app), worker thread will be waiting for `time.sleep` to finish and then for `service.get_pong` to finish
 - i. Sync operation blocks only the side thread, not the main one.
 - c. When `good_ping` finishes its work, server returns a response to the client
3. GET /perfect-pingFastAPI server receives a request and starts handling it
 - a. FastAPI awaits `asyncio.sleep(10)`
 - b. Event loop selects next tasks from the queue and works on them (e.g. accept new request, call db)
 - c. When `asyncio.sleep(10)` is done, servers goes to the next lines and awaits `service.async_get_pong`
 - d. Event loop selects next tasks from the queue and works on them (e.g. accept new request, call db)

e. When `service.async_get_pong` is done, server returns a response to the client

The second caveat is that operations that are non-blocking awaitables or are sent to the thread pool must be I/O intensive tasks (e.g. open file, db call, external API call).

- Awaiting CPU-intensive tasks (e.g. heavy calculations, data processing, video transcoding) is worthless since the CPU has to work to finish the tasks, while I/O operations are external and server does nothing while waiting for that operations to finish, thus it can go to the next tasks.
- Running CPU-intensive tasks in other threads also isn't effective, because of [GIL](#). In short, GIL allows only one thread to work at a time, which makes it useless for CPU tasks.
- If you want to optimize CPU intensive tasks you should send them to workers in another process.

1. **非阻塞的 `awaitable` 操作或发送到线程池的操作应该是 I/O 密集型任务：**在 Python 中，异步编程主要用于处理 I/O 密集型任务，如打开文件、数据库调用或外部 API 调用。当这些操作正在等待 I/O 完成时，主线程可以继续执行其他任务，从而提高程序的整体效率。

2. **在 CPU 密集型任务上使用异步编程或多线程效果并不好：**对于需要大量计算、数据处理或视频转码等 CPU 密集型任务，使用异步编程并无太大帮助，因为 CPU 必须完成这些任务，而在此期间，CPU 无法处理其他任务。同时，由于 Python 的全局解释器锁（GIL）的存在，多线程也无法有效地并行处理 CPU 密集型任务。GIL 确保了同一时间只有一个线程在执行，这使得 Python 的多线程对于 CPU 密集型任务来说并不高效。

3. **优化 CPU 密集型任务的方法是将它们发送到另一个进程的工作线程中：**如果你需要优化 CPU 密集型任务，一种有效的方法是使用多进程，而不是多线程。每个进程都有自己的 GIL，所以它们可以在不同的 CPU 核心上并行运行，从而有效地利用多核 CPU 的计算能力。

总的来说，这段话的意思是，在设计 Python 程序时，应该根据任务的类型（I/O 密集型还是 CPU 密集型）选择合适的并发模型（异步、多线程或多进程）。

FastAPI 是一个现代、快速（高性能）的 web 框架，它基于 Starlette（用于 web 部分）和 Pydantic（用于数据部分）。FastAPI 的一个重要特性是其异步处理能力，这意味着它可以同时处理多个请求，而不是顺序处理每个请求。在你的例子中，请求 A、B 和 C 应该是并发处理的，而不是串行处理的。

然而，由于 Python 的 GIL（全局解释器锁）限制，即使在多线程环境中，Python 代码仍然是串行执行的。这意味着在单个进程中，Python 代码的执行是阻塞的。但是，FastAPI 能够通过异步 I/O 来解决这个问题，并发处理多个请求。

在你的代码中，`asyncio.sleep(10)` 是一个异步操作，它不会阻塞事件循环，因此在等待这个操作完成的同时，FastAPI 可以处理其他的请求。然而，如果你在 FastAPI 应用中使用的是标准的 `time.sleep(10)`，那么这将会阻塞整个事件循环，使得 FastAPI 无法处理其他的请求。

如果你的请求仍然是串行处理的，可能的原因是你的 ASGI 服务器没有正确配置。FastAPI 依赖于 ASGI 服务器（如 Uvicorn 或 Hypercorn）来处理并发请求。你需要确保你的 ASGI 服务器配置为多工作进程或多线程模式。例如，如果你使用 Uvicorn，你可以使用 `--workers` 参数来设置工作进程的数量。

如果你已经正确配置了你的 ASGI 服务器，但是请求仍然是串行处理的，那么可能的原因是你的代码中存在阻塞操作，或者你的服务依赖于一些不能并发处理请求的外部服务。

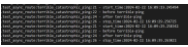
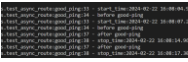
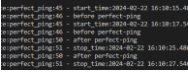
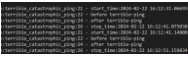
总的来说，如果你的 FastAPI 应用不能并发处理请求，你需要检查以下几点：

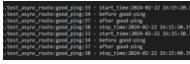
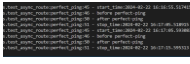

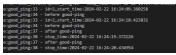
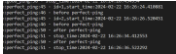
1. 确保你的 ASGI 服务器配置为多工作进程或多线程模式。
2. 检查你的代码中是否存在阻塞操作。
3. 检查你的服务是否依赖于一些不能并发处理请求的外部服务。

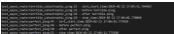
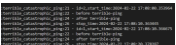
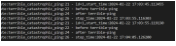


实验

连续发送两个请求，保证请求 1 还在执行，请求 2 已经发送

unicorn	route	是否两个浏览器	是否相同参	结果	结论
---------	-------	---------	-------	----	----

workers		(chrome 和 edge) 发起请求	数		
默认 (1 个)	terrible-ping	y	-	串行, 即 Fastapi 完成请求 1 后, 才开始接收请求 2 	不同浏览器发送同样请求时, terrible 会串行处理, 和最佳实践结论一致; 而 good 和 perfect 确实会并行处理;
	good-ping	y	-	并行, 即 Fastapi 处理请求 1 的同时, 可以接收请求 2 并处理 	
	perfect-ping	y	-	并行, 即 Fastapi 处理请求 1 的同时, 可以接收请求 2 并处理 	
	terrible	n	-	串行 	不论是哪种处理方式, 同一浏览器相同请求会
	good	n	-	串行	

					被串行处理，并且和浏览器无关，chrome 和 edge 结果都一样
	perfect	n	-	串行 	
	terrible	y	n	串行 	除了 terrible，同一浏览器不同请求会被并行处理，并且和浏览器无关，chrome 和 edge 结果都一样；如果请求一致则都为串行。
	good	y	n	并行 	
	perfect	y	n	并行 	

					<div>样的参数)</div> <div>会被阻塞</div> <div>同时先请求</div> <div>terrible 路</div> <div>由的请求会</div> <div>阻塞访问</div> <div>perfect 的</div> <div>路由请求 :</div> <div></div>
<div>2</div> <div></div>	terrible	y	n	串行 <div></div>	
	terrible	y	y	串行 <div></div>	
	terrilbe	n	y	串行 <div></div>	
	terrbile	n	n	串行 <div></div>	

综上所述：

terrible 无论是否增加 **workers** 都会阻塞后续的请求（访问自身路由以及其它路由）

如果是同样的请求（同一浏览器复制标签页、入参一致），不论是哪种方式都会阻塞变为串行处理（完全一样的请求会被阻塞，个人认为也是正常的，也能够防止恶意攻击），并且和浏览器无关

8. Custom base model from day 0.

Having a controllable global base model allows us to customize all the models within the app. For example, we could have a standard datetime format or add a super method for all subclasses of the base model.

9. Docs

1. Unless your API is public, hide docs by default. Show it explicitly on the selected envs only.
2. Help FastAPI to generate an easy-to-understand docsSet response_model, status_code, description, etc.
 - a. If models and statuses vary, use responses route attribute to add docs for different responses

```

from fastapi import APIRouter, status

router = APIRouter()

@router.post(
    "/endpoints",
    response_model=DefaultResponseModel, # default response
pydantic model
    status_code=status.HTTP_201_CREATED, # default status code
    description="Description of the well documented endpoint",
    tags=["Endpoint Category"],
    summary="Summary of the Endpoint",
    responses={
        status.HTTP_200_OK: {
            "model": OkResponse, # custom pydantic model for 200
response
            "description": "Ok Response",
        },
        status.HTTP_201_CREATED: {
            "model": CreatedResponse, # custom pydantic model
for 201 response
            "description": "Creates something from user request
",
        },
        status.HTTP_202_ACCEPTED: {
            "model": AcceptedResponse, # custom pydantic model
for 202 response
            "description": "Accepts request and handles it
later",
        },
    },
)

async def documented_route():
    pass

```

POST

/endpoints

Summary of the Endpoint

^

Description of the well documented endpoint

Parameters

Try it out

No parameters

Responses

Code	Description	Links
200	<div>Ok Response</div> <div>Media type</div> <div>application/json</div> <div>Controls Accept header.</div> <div>Example Value</div> <div>Schema</div> <pre>{ "status": "OK" }</pre>	No links
201	<div>Creates something from user request</div> <div>Media type</div> <div>application/json</div> <div>Example Value</div> <div>Schema</div>	No links

10. Use Pydantic's BaseSettings for configs

Pydantic gives a [powerful tool](#) to parse environment variables and process them with its validators.

```
from pydantic import AnyUrl, PostgresDsn
from pydantic_settings import BaseSettings # pydantic v2

class AppSettings(BaseSettings):
    class Config:
        env_file = ".env"
        env_file_encoding = "utf-8"
        env_prefix = "app_"

    DATABASE_URL: PostgresDsn
    IS_GOOD_ENV: bool = True
    ALLOWED_CORS_ORIGINS: set[AnyUrl]
```

11. SQLAlchemy: Set DB keys naming convention

Explicitly setting the indexes' namings according to your database's convention is preferable over sqlalchemy's.

根据数据库的约定明确设置索引的命名更可取。

```

from sqlalchemy import MetaData

POSTGRES_INDEXES_NAMING_CONVENTION = {
    "ix": "%(column_0_label)s_idx",
    "uq": "%(table_name)s_%(column_0_name)s_key",
    "ck": "%(table_name)s_%(constraint_name)s_check",
    "fk": "%(table_name)s_%(column_0_name)s_fkey",
    "pk": "%(table_name)s_pkey",
}

metadata =
MetaData(naming_convention=POSTGRES_INDEXES_NAMING_CONVENTION)

```

12. Migrations. Alembic.

1. Migrations must be static and revertable. If your migrations depend on dynamically generated data, then make sure the only thing that is dynamic is the data itself, not its structure.
2. Generate migrations with descriptive names & slugs. Slug is required and should explain the changes.
3. Set human-readable file template for new migrations. We use `*date*_slug*.py` pattern, e.g. [2022-08-24_post_content_idx.py](#)

13. Set DB naming convention

Being consistent with names is important. Some rules we followed:

1. lower_case_snake
2. singular form (e.g. post, post_like, user_playlist)
3. group similar tables with module prefix, e.g. payment_account, payment_bill, post, post_like
4. stay consistent across tables, but concrete namings are ok, e.g. use profile_id in all tables, but if some of them need only profiles that are creators, use creator_id
 - a. use post_id for all abstract tables like post_like, post_view, but use concrete naming in relevant modules like course_id in chapters.course_id

5. `_at` suffix for datetime

6. `_date` suffix for date

14. Set tests client async from day 0

Writing integration tests with DB will most likely lead to messed up event loop errors in the future. Set the async test client immediately, e.g. [async_asgi_testclient](#) or [httpx](#)

15. BackgroundTasks > `asyncio.create_task`

BackgroundTasks can [effectively run](#) both blocking and non-blocking I/O operations the same way FastAPI handles blocking routes (sync tasks are run in a threadpool, while async tasks are awaited later)

- Don't lie to the worker and don't mark blocking I/O operations as async
- Don't use it for heavy CPU intensive tasks.

这段话主要在讨论 FastAPI 的 BackgroundTasks 和如何在其上使用阻塞和非阻塞 I/O 操作。这里分三个部分进行解释：

1. **BackgroundTasks 可以同时处理阻塞和非阻塞 I/O 操作：**FastAPI 的 BackgroundTasks 类允许你在请求处理完成后在后台运行一些额外的任务。这些任务可以是阻塞的，也可以是非阻塞的。对于阻塞任务，BackgroundTasks 会将它们发送到一个线程池中运行，以避免阻塞主线程。对于非阻塞任务，BackgroundTasks 会在主线程空闲时进行 `await`。

2. **不要将阻塞 I/O 操作标记为异步：**在 Python 中，异步函数通常用于处理 I/O 密集型任务，而不是 CPU 密集型任务。如果你将一个阻塞 I/O 操作标记为异步，那么你就在向工作线程“撒谎”，这可能会导致程序的效率降低。因此，当你在编写异步函数时，应该确保它们真正做的是非阻塞的 I/O 操作。

3. **不要在 BackgroundTasks 中运行 CPU 密集型任务：**如之前所述，BackgroundTasks 主要用于处理 I/O 密集型任务，而不是 CPU 密集型任务。如果你在 BackgroundTasks 中运行一个 CPU 密集型任务，那么这可能会阻塞主线程，从而降低程序的效率。对于 CPU 密集型任务，你应该考虑使用其他的并发模型，如多进程。

总的来说，这段话的意思是，当你在使用 FastAPI 的 BackgroundTasks 时，应该确保你正确地使用

了阻塞和非阻塞 I/O 操作，并避免在其中运行 CPU 密集型任务。

```
from fastapi import APIRouter, BackgroundTasks
from pydantic import UUID4

from src.notifications import service as notifications_service

router = APIRouter()

@router.post("/users/{user_id}/email")
async def send_user_email(worker: BackgroundTasks, user_id:
UUID4):
    """Send email to user"""
    worker.add_task(notifications_service.send_email, user_id) #
    send email after responding client
    return {"status": "ok"}
```

16. Typing is important

FastAPI, Pydantic, and modern IDEs encourage to take use of type hints.

17. Save files in chunks.

Don't hope your clients will send small files.

```
import aiofiles
from fastapi import UploadFile

DEFAULT_CHUNK_SIZE = 1024 * 1024 * 50 # 50 megabytes

async def save_video(video_file: UploadFile):
    async with aiofiles.open("/file/path/name.mp4", "wb") as f:
        while chunk := await video_file.read(DEFAULT_CHUNK_SIZE):
            await f.write(chunk)
```

18. Be careful with dynamic pydantic fields (Pydantic v1)

If you have a pydantic field that can accept a union of types, be sure the validator explicitly

knows the difference between those types.

当你在 Pydantic 模型中使用 Union 时，你应该确保你的验证器能够明确地区分不同的类型。你可能需要定义一个自定义的验证器来处理这种情况，或者更改类型的顺序以确保它们按照你期望的方式进行验证。

总的来说，这句话的意思是：当你在 Pydantic 模型中定义一个可以接受多种类型的字段时，你需要确保你的验证器能够明确地区分这些类型，以避免出现意外的验证结果。

示例：

```
from pydantic import BaseModel

class Article(BaseModel):
    text: str | None
    extra: str | None

class Video(BaseModel):
    video_id: int
    text: str | None
    extra: str | None

class Post(BaseModel):
    content: Article | Video

post = Post(content={"video_id": 1, "text": "text"})
print(type(post.content))
# OUTPUT: Article
# Article is very inclusive and all fields are optional, allowing
any dict to become valid
```

解决办法：

1. Validate input has only allowed valid fields and raise error if unknowns are provided

```

from pydantic import BaseModel, Extra

class Article(BaseModel):
    text: str | None
    extra: str | None

    class Config:
        extra = Extra.forbid

class Video(BaseModel):
    video_id: int
    text: str | None
    extra: str | None

    class Config:
        extra = Extra.forbid

class Post(BaseModel):
    content: Article | Video

```

2. Use Pydantic's Smart Union (>v1.9, <2.0) if fields are simple

It's a good solution if the fields are simple like int or bool, but it doesn't work for complex fields like classes.

3. Fast Workaround

Order field types properly: from the most strict ones to loose ones.

19. SQL-first, Pydantic-second

- Usually, database handles data processing much faster and cleaner than CPython will ever do.
- It's preferable to do all the complex joins and simple data manipulations with SQL.
- It's preferable to aggregate JSONs in DB for responses with nested objects.

20. Validate hosts, if users can send publicly available URLs

For example, we have a specific endpoint which:

1. accepts media file from the user,
2. generates unique url for this file,
3. returns url to user, which they will use in other endpoints like PUT /profiles/me, POST /posts
 - a. these endpoints accept files only from whitelisted hosts
4. uploads file to AWS with this name and matching URL.

If we don't whitelist URL hosts, then bad users will have a chance to upload dangerous links.

```

from pydantic import AnyUrl, BaseModel

ALLOWED_MEDIA_URLS = {"mysite.com", "mysite.org"}

class CompanyMediaUrl(AnyUrl):
    @classmethod
    def validate_host(cls, parts: dict) -> tuple[str, str, str,
bool]: # pydantic v1
        """Extend pydantic's AnyUrl validation to whitelist URL
hosts."""
        host, tld, host_type, rebuild =
super().validate_host(parts)
        if host not in ALLOWED_MEDIA_URLS:
            raise ValueError(
                "Forbidden host url. Upload files only to
internal services."
            )

        return host, tld, host_type, rebuild

class Profile(BaseModel):
    avatar_url: CompanyMediaUrl # only whitelisted urls for
avatar

```

21. Raise a ValueError in custom pydantic validators, if schema directly faces the client

It will return a nice detailed response to users.

```

# src.profiles.schemas
from pydantic import BaseModel, validator

class ProfileCreate(BaseModel):
    username: str

    @validator("username") # pydantic v1
    def validate_bad_words(cls, username: str):
        if username == "me":
            raise ValueError("bad username, choose another")

        return username

# src.profiles.routes
from fastapi import APIRouter

router = APIRouter()

@router.post("/profiles")
async def get_creator_posts(profile_data: ProfileCreate):
    pass

```

22. FastAPI converts Pydantic objects to dict, then to Pydantic object, then to JSON

If you think you can return Pydantic object that matches your route's `response_model` to make some optimizations, then it's wrong.

FastAPI firstly converts that pydantic object to dict with its `jsonable_encoder`, then validates data with your `response_model`, and only then serializes your object to JSON.

在 FastAPI 中，你可以使用 `response_model` 参数来指定路由的响应模型。这个响应模型通常是一个 Pydantic 模型，用于定义响应数据的结构和类型。当你的路由处理函数返回一个响应时，FastAPI 会使用这个响应模型来验证和转换响应数据。

你可能会认为，如果你的路由处理函数直接返回一个与 `response_model` 匹配的 Pydantic 对象，那么 FastAPI 就可以省去验证和转换的步骤，从而优化性能。然而，这其实是错误的。

当你的路由处理函数返回一个 Pydantic 对象时，FastAPI 会首先使用其 `jsonable_encoder` 函数将这个对象转换为一个字典。然后，FastAPI 会使用你指定的 `response_model` 来验证这个字典。最后，FastAPI 会将验证后的数据序列化为 JSON 格式，然后将其发送给客户端。

所以，即使你的路由处理函数返回一个与 `response_model` 完全匹配的 Pydantic 对象，FastAPI 仍然会进行数据的验证和转换。这意味着你不能通过直接返回 Pydantic 对象来优化性能。

总的来说，这段话的意思是：在 FastAPI 中，你不能通过直接返回与 `response_model` 匹配的 Pydantic 对象来优化性能，因为 FastAPI 总是会对响应数据进行验证和转换。

23. If you must use sync SDK, then run it in a thread pool.

If you must use a library to interact with external services, and it's not async, then make the HTTP calls in an external worker thread.

这段话在讨论 Python 中的同步（sync）和异步（async）编程模式，以及如何在异步环境中使用同步软件开发包（SDK）或库。

在 Python 中，异步编程是一种编程模式，它允许程序在等待某个操作完成（如 I/O 操作）时继续执行其他任务。这对于 I/O 密集型任务（如网络请求或文件读写）非常有用，因为这些操作通常需要花费一些时间来完成，而在此期间，CPU 可以做其他的事情。

然而，并非所有的库都支持异步编程。有些库，特别是一些旧的或者专门针对 CPU 密集型任务的库，可能只提供同步的接口。这种情况下，如果你在异步环境中直接调用这些库的函数，可能会阻塞整个事件循环，从而降低程序的效率。

所以，如果你必须在异步环境中使用一个同步的 SDK 或库，那么你应该在一个线程池中运行它。线程池是一种并发模型，它允许你在多个线程中运行任务。当你在线程池中运行一个任务时，这个任务会在一个新的线程中执行，从而避免阻塞主线程。

总的来说，这段话的意思是：如果你必须在异步环境中使用一个同步的 SDK 或库，那么你应该在一个线程池中运行它，以避免阻塞主线程。

For a simple example, we could use our well-known `run_in_threadpool` from `starlette`.

```
from fastapi import FastAPI
from fastapi.concurrency import run_in_threadpool
from my_sync_library import SyncAPIClient

app = FastAPI()

@app.get("/")
async def call_my_sync_library():
    my_data = await service.get_my_data()

    client = SyncAPIClient()
    await run_in_threadpool(client.make_request, data=my_data)
```

24. Use linters (black, ruff)

With linters, you can forget about formatting the code and focus on writing the business logic.