
FastAPI Boilerplate

Features

- Async SQLAlchemy session
- Custom user class
- Dependencies for specific permissions
- Celery
- Dockerize(Hot reload)
- Event dispatcher
- Cache

Run

Launch docker

```
1 > docker-compose -f docker/docker-compose.yml up
```

Install dependency

```
1 > poetry shell
2 > poetry install
```

Apply alembic revision

```
1 > alembic upgrade head
```

Run server

```
1 > python3 main.py --env local|dev|prod --debug
```

Run test codes

```
1 > make test
```

Make coverage report

```
1 > make cov
```

Formatting

```
1 > pre-commit
```

SQLAlchemy for asyncio context

```
1 from core.db import Transactional, session
2
3
4 @Transactional()
5 async def create_user(self):
6     session.add(User(email="padocon@naver.com"))
```

Do not use explicit `commit()`. `Transactional` class automatically do.

Query with `asyncio.gather()`

When executing queries concurrently through `asyncio.gather()`, you must use the `session_factory` context manager rather than the globally used session.

```
1 from core.db import session_factory
2
3
4 async def get_by_id(self, *, user_id) -> User:
5     stmt = select(User)
6     async with session_factory() as read_session:
7         return await read_session.execute(query).scalars().first()
8
9
10 async def main() -> None:
11     user_1, user_2 = await asyncio.gather(
12         get_by_id(user_id=1),
13         get_by_id(user_id=2),
14     )
```

If you do not use a database connection like `session.add()`, it is recommended to use a globally provided session.

Multiple databases

Go to `core/config.py` and edit `WRITER_DB_URL` and `READER_DB_URL` in the config class.

If you need additional logic to use the database, refer to the `get_bind()` method of `RoutingClass`.

Custom user for authentication

```
1 from fastapi import Request
2
3
4 @home_router.get("/")
5 def home(request: Request):
6     return request.user.id
```

Note. you have to pass jwt token via header like `Authorization: Bearer 1234`

Custom user class automatically decodes header token and store user information into `request.user`

If you want to modify custom user class, you have to update below files.

1. `core/fastapi/schemas/current_user.py`
2. `core/fastapi/middlewares/authentication.py`

CurrentUser

```
1 class CurrentUser(BaseModel):
2     id: int = Field(None, description="ID")
```

Simply add more fields based on your needs.

AuthBackend

```
1 current_user = CurrentUser()
```

After line 18, assign values that you added on `CurrentUser`.

Top-level dependency

Note. Available from version 0.62 or higher.

Set a callable function when initialize FastAPI() app through `dependencies` argument.

Refer `Logging` class inside of `core/fastapi/dependencies/logging.py`

Dependencies for specific permissions

Permissions `IsAdmin`, `IsAuthenticated`, `AllowAll` have already been implemented.

```
1 from core.fastapi.dependencies import (
2     PermissionDependency,
3     IsAdmin,
4 )
5
6
7 user_router = APIRouter()
8
9
10 @user_router.get(
11     "",
12     response_model=List[GetUserListResponseSchema],
13     response_model_exclude={"id"},
14     responses={"400": {"model": ExceptionResponseSchema}},
15     dependencies=[Depends(PermissionDependency([IsAdmin]))], # HERE
16 )
17 async def get_user_list(
18     limit: int = Query(10, description="Limit"),
19     prev: int = Query(None, description="Prev ID"),
20 ):
21     pass
```

Insert permission through `dependencies` argument.

If you want to make your own permission, inherit `BasePermission` and implement `has_permission()` function.

Note. In order to use swagger's `authorize` function, you must put `PermissionDependency` as an argument of `dependencies`.

Event dispatcher

Refer the README of <https://github.com/teamhide/fastapi-event>

Cache

Caching by prefix

```
1 from core.helpers.cache import Cache
2
3
4 @Cache.cached(prefix="get_user", ttl=60)
5 async def get_user():
6     ...
```

Caching by tag

```
1 from core.helpers.cache import Cache, CacheTag
2
3
4 @Cache.cached(tag=CacheTag.GET_USER_LIST, ttl=60)
5 async def get_user():
6     ...
```

Use the `Cache` decorator to cache the return value of a function.

Depending on the argument of the function, caching is stored with a different value through internal processing.

Custom Key builder

```
1 from core.helpers.cache.base import BaseKeyMaker
2
3
4 class CustomKeyMaker(BaseKeyMaker):
5     async def make(self, function: Callable, prefix: str) -> str:
6         ...
```

If you want to create a custom key, inherit the `BaseKeyMaker` class and implement the `make()` method.

Custom Backend

```
1 from core.helpers.cache.base import BaseBackend
2
3
4 class RedisBackend(BaseBackend):
5     async def get(self, key: str) -> Any:
6         ...
7
8     async def set(self, response: Any, key: str, ttl: int = 60) -> None
9         :
```

```
9         ...
10
11     async def delete_startswith(self, value: str) -> None:
12         ...
```

If you want to create a custom key, inherit the BaseBackend class and implement the `get()`, `set()`, `delete_startswith()` method.

Pass your custom backend or keymaker as an argument to `init`. (`/app/server.py`)

```
1 def init_cache() -> None:
2     Cache.init(backend=RedisBackend(), key_maker=CustomKeyMaker())
```

Remove all cache by prefix/tag

```
1 from core.helpers.cache import Cache, CacheTag
2
3
4 await Cache.remove_by_prefix(prefix="get_user_list")
5 await Cache.remove_by_tag(tag=CacheTag.GET_USER_LIST)
```