Day 02 Concepts -

* How to read and write in a Postgres DB.
* How to access JWS Authentication, refresh Token for API Calls

\*\* Validate your learning from DB integration from the course

Use the below Open Source example - https://github.com/mfreeborn/fastapi-sqlalchemy (name of the repository is fastapi-sqlalchemy)

1. Connect to Test DB, add a single Table Employee  - Write an API to store employee records in the DB
2. Write a Get API to implement paging - Page size is 2

Implement JWS Authentication as shown here - https://github.com/dmontagu/fastapi-auth

this information is in the day 2 concepts and inside the git repository that also has information about the implementation of JWS authentication. analyze all this information also and then give me the full logical code. day 2 information inside the git repository.

I am also attaching that information inside that Github repository.

analyze all information in that repository and there what are information is required to complete the day 2 concepts.

give me the file structure and full detailed codes and their logic.

use

inside the git repository there are fast-auth folderss. inside that folder there are features, orm, setting, setup and util folder are present and also api\_model.py file.

code inside the api\_model.py file

from functools import partial

from pydantic import BaseConfig, BaseModel

from fastapi\_auth.fastapi\_util.util.camelcase import snake2camel

class APIModel(BaseModel):

class Config(BaseConfig):

orm\_mode = True

allow\_population\_by\_field\_name = True

alias\_generator = partial(snake2camel, start\_lower=True)

class APIMessage(APIModel):

detail: str

inside the features folder, there are program files.

cbv.py and inferring\_routher.py files code.

code inside cbv.py file.

import inspect

from typing import Any, Callable, List, Type, TypeVar, Union, get\_type\_hints

from fastapi import APIRouter, Depends

from pydantic.typing import is\_classvar

from starlette.routing import Route, WebSocketRoute

T = TypeVar("T")

CBV\_CLASS\_KEY = "\_\_cbv\_class\_\_"

def cbv(router: APIRouter) -> Callable[[Type[T]], Type[T]]:

def decorator(cls: Type[T]) -> Type[T]:

return \_cbv(router, cls)

return decorator

def \_cbv(router: APIRouter, cls: Type[T]) -> Type[T]:

*init*cbv(cls)

cbv\_router = APIRouter()

functions = inspect.getmembers(cls, inspect.isfunction)

routes\_by\_endpoint = {

route.endpoint: route for route in router.routes if isinstance(route, (Route, WebSocketRoute))

}

for \_, func in functions:

route = routes\_by\_endpoint.get(func)

if route is None:

continue

router.routes.remove(route)

*update*cbv\_route\_endpoint\_signature(cls, route)

cbv\_router.routes.append(route)

router.include\_router(cbv\_router)

return cls

def *update*cbv\_route\_endpoint\_signature(cls: Type[Any], route: Union[Route, WebSocketRoute]) -> None:

old\_endpoint = route.endpoint

old\_signature = inspect.signature(old\_endpoint)

old\_parameters: List[inspect.Parameter] = list(old\_signature.parameters.values())

old\_first\_parameter = old\_parameters[0]

new\_first\_parameter = old\_first\_parameter.replace(default=Depends(cls))

new\_parameters = [new\_first\_parameter] + [

parameter.replace(kind=inspect.Parameter.KEYWORD\_ONLY) for parameter in old\_parameters[1:]

]

new\_signature = old\_signature.replace(parameters=new\_parameters)

setattr(route.endpoint, "\_\_signature\_\_", new\_signature)

def *init*cbv(cls: Type[Any]) -> None:

if getattr(cls, CBV\_CLASS\_KEY, False): # pragma: no cover

return # Already initialized

old\_init: Callable[..., Any] = cls.\_\_init\_\_

old\_signature = inspect.signature(old\_init)

old\_parameters = list(old\_signature.parameters.values())[1:] # drop self parameter

new\_parameters = [

x for x in old\_parameters if x.kind not in (inspect.Parameter.VAR\_POSITIONAL, inspect.Parameter.VAR\_KEYWORD)

]

dependency\_names: List[str] = []

for name, hint in get\_type\_hints(cls).items():

if is\_classvar(hint):

continue

parameter\_kwargs = {}

parameter\_kwargs["default"] = getattr(cls, name, Ellipsis)

dependency\_names.append(name)

new\_parameters.append(

inspect.Parameter(name=name, kind=inspect.Parameter.KEYWORD\_ONLY, annotation=hint, \*\*parameter\_kwargs)

)

new\_signature = old\_signature.replace(parameters=new\_parameters)

def new\_init(self: Any, *args: Any,* \*kwargs: Any) -> None:

for dep\_name in dependency\_names:

dep\_value = kwargs.pop(dep\_name)

setattr(self, dep\_name, dep\_value)

old\_init(self, *args,* \*kwargs)

setattr(cls, "\_\_signature\_\_", new\_signature)

setattr(cls, "\_\_init\_\_", new\_init)

setattr(cls, CBV\_CLASS\_KEY, True)

code inside inferring\_router.py.

from typing import TYPE\_CHECKING, Any, Callable, get\_type\_hints

from fastapi import APIRouter

class InferringRouter(APIRouter):

if not TYPE\_CHECKING:

def add\_api\_route(self, path: str, endpoint: Callable[..., Any], \*\*kwargs: Any) -> None:

if kwargs.get("response\_model") is None:

kwargs["response\_model"] = get\_type\_hints(endpoint).get("return")

return super().add\_api\_route(path, endpoint, \*\*kwargs)

else: # pragma: no cover

pass

orm folder and the inside files base.py, columns.py and guid\_type.py

base.py file and there code.

from typing import TYPE\_CHECKING, Any, Dict, TypeVar

import sqlalchemy as sa

from sqlalchemy.ext.declarative import DeclarativeMeta, declarative\_base, declared\_attr

from sqlalchemy.orm import Session

from sqlalchemy.sql.base import ImmutableColumnCollection

from fastapi\_auth.fastapi\_util.util.camelcase import camel2snake

T = TypeVar("T", bound="CustomBase")

MYPY = False

class CustomMeta(DeclarativeMeta):

**table**: sa.Table

@property

def columns(cls) -> ImmutableColumnCollection:

return cls.\_\_table\_\_.columns

class CustomBase:

**table**: sa.Table

if TYPE\_CHECKING:

**tablename**: str

else:

@declared\_attr

def **tablename**(cls) -> str:

return camel2snake(cls.\_\_name\_\_)

def dict(self) -> Dict[str, Any]:

data = {key: getattr(self, key) for key in self.\_\_table\_\_.c.keys()}

return data

*Base = declarative*base(cls=CustomBase, metaclass=CustomMeta)

if TYPE\_CHECKING:

# This is necessary for pycharm, but not mypy

class Base(\_Base, CustomBase, metaclass=CustomMeta):

**table**: sa.Table

\_\_tablename\_: str

metadata: sa.MetaData

columns: ImmutableColumnCollection

if not MYPY: # pragma: no cover

# Suppress pycharm kwargs warnings

def **init**(self, \*\*kwargs: Any) -> None:

pass

def dict(self) -> Dict[str, Any]:

...

else:

exec("Base = \_Base") # Need to hide from PyCharm

S = TypeVar("S", bound="Base")

def add\_base(session: Session, item: S) -> S:

session.add(item)

session.commit()

return item

code inside columns.py file.

from datetime import datetime

from typing import Any, Dict, Optional, Type, TypeVar, Union, overload

from uuid import UUID, uuid4

import sqlalchemy as sa

from sqlalchemy.dialects.postgresql import JSONB

from typing\_extensions import Literal

from fastapi\_auth.fastapi\_util.orm.guid\_type import GUID

from fastapi\_auth.fastapi\_util.settings.database\_settings import DatabaseBackend, get\_database\_settings

IdentifierT = TypeVar("IdentifierT", bound=UUID)

@overload

def pk\_column(id\_type: Type[IdentifierT]) -> "sa.Column[IdentifierT]":

...

@overload

def pk\_column(id\_type: None = None) -> "sa.Column[UUID]":

...

def pk\_column(id\_type: Optional[Type[IdentifierT]] = None) -> "Union[sa.Column[IdentifierT], sa.Column[UUID]]":

"""

The server-default value should be updated in the metadata later

"""

using\_postgres = get\_database\_settings().backend == DatabaseBackend.postgresql

default\_kwargs: Dict[str, Any] = {"default": uuid4} if not using\_postgres else {

"server\_default": sa.text("gen\_random\_uuid()")

}

return sa.Column(GUID, primary\_key=True, index=True, \*\*default\_kwargs)

@overload

def fk\_column(

column: Union[str, "sa.Column[IdentifierT]"],

nullable: Literal[True],

index: bool = False,

primary\_key: bool = False,

unique: bool = False,

) -> "sa.Column[Optional[IdentifierT]]":

...

@overload

def fk\_column(

column: Union[str, "sa.Column[IdentifierT]"],

nullable: Literal[False] = False,

index: bool = False,

primary\_key: bool = False,

unique: bool = False,

) -> "sa.Column[IdentifierT]":

...

def fk\_column(

column: Union[str, "sa.Column[IdentifierT]"],

nullable: bool = False,

index: bool = False,

primary\_key: bool = False,

unique: bool = False,

) -> "Union[sa.Column[IdentifierT], sa.Column[Optional[IdentifierT]]]":

return sa.Column( # type: ignore

GUID,

sa.ForeignKey(column, ondelete="CASCADE"),

index=index,

nullable=nullable,

primary\_key=primary\_key,

unique=unique,

)

def json\_column(\*, nullable: bool) -> "sa.Column[Dict[str, Any]]":

using\_postgres = get\_database\_settings().backend == DatabaseBackend.postgresql

column\_type = JSONB() if using\_postgres else sa.JSON()

return sa.Column(column\_type, nullable=nullable) # type: ignore

def created\_at\_column() -> "sa.Column[datetime]":

return sa.Column(sa.DateTime(timezone=True), server\_default=sa.func.now(), nullable=False)

def updated\_at\_column() -> "sa.Column[datetime]":

return sa.Column(sa.DateTime(timezone=True), server\_default=sa.func.now(), onupdate=sa.func.now(), nullable=False)

guid\_type.py file and there inside code.

import uuid

from typing import no\_type\_check

from sqlalchemy.dialects.postgresql.base import UUID

from sqlalchemy.sql.sqltypes import CHAR

from sqlalchemy.sql.type\_api import TypeDecorator

class GUID(TypeDecorator): # type: ignore

"""

Platform-independent GUID type.

Uses PostgreSQL's UUID type, otherwise uses CHAR(32), storing as stringified hex values.

Taken from SQLAlchemy docs: https://docs.sqlalchemy.org/en/13/core/custom\_types.html#backend-agnostic-guid-type

"""

impl = CHAR

@no\_type\_check

def **init**(self, *args,* \*kwargs):

super().\_\_init\_\_(\*args, \*\*kwargs)

@no\_type\_check

def load\_dialect\_impl(self, dialect):

if dialect.name == "postgresql":

return dialect.type\_descriptor(UUID())

else:

return dialect.type\_descriptor(CHAR(32))

@no\_type\_check

def process\_bind\_param(self, value, dialect):

if value is None:

return value

elif dialect.name == "postgresql":

return str(value)

else:

if not isinstance(value, uuid.UUID):

return "%.32x" % uuid.UUID(value).int

else:

# hexstring

return "%.32x" % value.int

@no\_type\_check

def process\_result\_value(self, value, dialect):

if value is None:

return value

else:

if not isinstance(value, uuid.UUID):

value = uuid.UUID(value)

return value

inside the setting folder ther are also programing file known as api\_settings.py, base\_api\_settings.py, and database\_setting.py files.

code inside the api\_settings.py file is

from functools import lru\_cache

from typing import Any, Dict

from fastapi\_auth.fastapi\_util.settings.base\_api\_settings import BaseAPISettings

class APISettings(BaseAPISettings):

# fastapi.applications.FastAPI initializer kwargs

debug: bool = False

docs\_url: str = "/docs"

openapi\_prefix: str = ""

openapi\_url: str = "/openapi.json"

redoc\_url: str = "/redoc"

title: str = "Fast API"

version: str = "0.1.0"

# Custom settings

disable\_docs: bool = False

disable\_superuser\_dependency: bool = False

include\_admin\_routes: bool = False

main\_router\_prefix: str = "/api/v1"

@property

def fastapi\_kwargs(self) -> Dict[str, Any]:

fastapi\_kwargs: Dict[str, Any] = {

"debug": self.debug,

"docs\_url": self.docs\_url,

"openapi\_prefix": self.openapi\_prefix,

"openapi\_url": self.openapi\_url,

"redoc\_url": self.redoc\_url,

"title": self.title,

"version": self.version,

}

if self.disable\_docs:

fastapi\_kwargs.update({"docs\_url": None, "openapi\_url": None, "redoc\_url": None})

return fastapi\_kwargs

class Config:

env\_prefix = "api\_"

@lru\_cache()

def get\_api\_settings() -> APISettings:

return APISettings()

code inside base\_api\_settings.py file.

from pydantic import BaseSettings

class BaseAPISettings(BaseSettings):

class Config:

env\_prefix = ""

arbitrary\_types\_allowed = True

validate\_assignment = True

database\_setting.py file code is

from enum import auto

from functools import lru\_cache

from typing import Any, Dict, Optional

import sqlalchemy as sa

from pydantic import validator

from fastapi\_auth.fastapi\_util.settings.base\_api\_settings import BaseAPISettings

from fastapi\_auth.fastapi\_util.util.enums import StrEnum

class DatabaseBackend(StrEnum):

postgresql = auto()

sqlite = auto()

@staticmethod

def from\_engine(engine: sa.engine.Engine) -> "DatabaseBackend":

return DatabaseBackend(engine.dialect.name)

class DatabaseSettings(BaseAPISettings):

backend: DatabaseBackend = None # type: ignore

user: Optional[str]

password: Optional[str]

host: Optional[str]

db: Optional[str]

sqlalchemy\_uri: str = None # type: ignore

log\_sqlalchemy\_sql\_statements: bool = False

min\_size: int = 10

max\_size: int = 10

force\_rollback: bool = False

@validator("sqlalchemy\_uri", pre=True, always=True)

def validate\_sqlalchemy\_uri(cls, v: Optional[str], values: Dict[str, Any]) -> str:

if v is None:

backend = values.get("backend")

backend = backend.value if backend is not None else None

user = values["user"]

password = values["password"]

host = values["host"]

db = values["db"]

v = f"{backend}://{user}:{password}@{host}/{db}"

return v

class Config:

env\_prefix = "db\_"

@lru\_cache()

def get\_database\_settings() -> DatabaseSettings:

return DatabaseSettings()

setup folder and other files are initialize.py, setup\_api.py, and setup\_database.py

initialize.py file and there code

import sqlalchemy as sa

from fastapi import FastAPI

from fastapi\_auth.fastapi\_util.orm.base import Base

from fastapi\_auth.fastapi\_util.setup.setup\_database import setup\_database, setup\_database\_metadata

from fastapi\_auth.fastapi\_util.util.session import get\_engine

def initialize\_database(engine: sa.engine.Engine) -> None:

setup\_database(engine)

def get\_configured\_metadata(\_app: FastAPI) -> sa.MetaData:

"""

This function accepts the app instance as an argument purely as a check to ensure that all resources

the app depends on have been imported.

In particular, this ensures the sqlalchemy metadata is populated.

"""

engine = get\_engine()

setup\_database(engine)

setup\_database\_metadata(Base.metadata, engine)

return Base.metadata

setup\_api.py file and other code is

from fastapi import FastAPI

from fastapi.routing import APIRoute

def setup\_openapi(app: FastAPI) -> None:

""" Simplify operation IDs so that generated clients have simpler api function names """

for route in app.routes:

if isinstance(route, APIRoute):

route.operation\_id = route.name

setup\_database.py file and other code is

import sqlalchemy as sa

from fastapi\_auth.fastapi\_util.orm.guid\_type import GUID

from fastapi\_auth.fastapi\_util.settings.database\_settings import DatabaseBackend

def setup\_database(engine: sa.engine.Engine) -> None:

setup\_guids(engine)

def setup\_guids(engine: sa.engine.Engine) -> None:

"""

Set up UUID generation using the uuid-ossp extension for postgres

"""

database\_backend = DatabaseBackend.from\_engine(engine)

# TODO: Add some way to run postgres-specific tests

if database\_backend == DatabaseBackend.postgresql: # pragma: no cover

# noinspection SqlDialectInspection,SqlNoDataSourceInspection

uuid\_generation\_setup\_query = 'create EXTENSION if not EXISTS "pgcrypto"'

engine.execute(uuid\_generation\_setup\_query)

def setup\_database\_metadata(metadata: sa.MetaData, engine: sa.engine.Engine) -> None:

setup\_guid\_server\_defaults(metadata, engine)

def setup\_guid\_server\_defaults(metadata: sa.MetaData, engine: sa.engine.Engine) -> None:

database\_backend = DatabaseBackend.from\_engine(engine)

guid\_server\_defaults = {

DatabaseBackend.postgresql: "gen\_random\_uuid()",

DatabaseBackend.sqlite: "(lower(hex(randomblob(16))))",

}

for table in metadata.tables.values():

if len(table.primary\_key.columns) != 1:

continue

for column in table.primary\_key.columns:

if type(column.type) is GUID:

column.server\_default = sa.DefaultClause(sa.text(guid\_server\_defaults[database\_backend]))

util folder and other files are camelcase.py, enums.py, session.py, tasks.py, and timing.py

code inside camelcase.py file

import re

def snake2camel(snake: str, start\_lower: bool = False) -> str:

camel = snake.title()

camel = re.sub("([0-9A-Za-z])\_(?=[0-9A-Z])", lambda m: m.group(1), camel)

if start\_lower:

camel = re.sub("(^\_\*[A-Z])", lambda m: m.group(1).lower(), camel)

return camel

def camel2snake(camel: str) -> str:

snake = re.sub(r"([a-zA-Z])([0-9])", lambda m: f"{m.group(1)}\_{m.group(2)}", camel)

snake = re.sub(r"([a-z0-9])([A-Z])", lambda m: f"{m.group(1)}\_{m.group(2)}", snake)

return snake.lower()

enums.py file code is

from enum import Enum

from fastapi\_auth.fastapi\_util.util.camelcase import snake2camel

class StrEnum(str, Enum):

# noinspection PyMethodParameters

def *generate*next\_value\_(name, start, count, last\_values) -> str: # type: ignore

"""

Uses the name as the automatic value, rather than an integer

See https://docs.python.org/3/library/enum.html#using-automatic-values for reference

"""

return name

class CamelStrEnum(str, Enum):

# noinspection PyMethodParameters

def *generate*next\_value\_(name, start, count, last\_values) -> str: # type: ignore

"""

Uses the camelCase name as the automatic value, rather than an integer

See https://docs.python.org/3/library/enum.html#using-automatic-values for reference

"""

return snake2camel(name, start\_lower=True)

session.py file code is

import logging

from contextlib import contextmanager

from functools import lru\_cache

from typing import Iterator, Optional

import sqlalchemy as sa

from sqlalchemy.orm import Session

from fastapi\_auth.fastapi\_util.settings.database\_settings import DatabaseBackend, get\_database\_settings

@lru\_cache()

def get\_engine() -> sa.engine.Engine:

db\_settings = get\_database\_settings()

uri = db\_settings.sqlalchemy\_uri

log\_sqlalchemy\_sql\_statements = db\_settings.log\_sqlalchemy\_sql\_statements

database\_backend = db\_settings.backend

return get\_new\_engine(uri, log\_sqlalchemy\_sql\_statements, database\_backend)

@lru\_cache()

def get\_sessionmaker() -> sa.orm.sessionmaker:

return get\_sessionmaker\_for\_engine(get\_engine())

def get\_new\_engine(

uri: str,

log\_sqlalchemy\_sql\_statements: bool = False,

database\_backend: DatabaseBackend = DatabaseBackend.postgresql,

) -> sa.engine.Engine:

if log\_sqlalchemy\_sql\_statements:

logging.getLogger("sqlalchemy.engine").setLevel(logging.INFO)

else:

logging.getLogger("sqlalchemy.engine").setLevel(logging.ERROR)

kwargs = {}

if database\_backend == DatabaseBackend.sqlite:

kwargs.update({"connect\_args": {"check\_same\_thread": False}})

return sa.create\_engine(uri, pool\_pre\_ping=True, \*\*kwargs)

def get\_sessionmaker\_for\_engine(engine: sa.engine.Engine) -> sa.orm.sessionmaker:

return sa.orm.sessionmaker(autocommit=False, autoflush=False, bind=engine)

def get\_session() -> sa.orm.Session:

return get\_sessionmaker()()

@contextmanager

def context\_session(engine: Optional[sa.engine.Engine] = None) -> Iterator[Session]:

yield from *get*db(engine)

def get\_db() -> Iterator[Session]:

"""

Intended for use as a FastAPI dependency

"""

yield from *get*db()

def *get*db(engine: Optional[sa.engine.Engine] = None) -> Iterator[Session]:

if engine is None:

session = get\_session()

else:

session = get\_sessionmaker\_for\_engine(engine)()

try:

yield session

session.commit()

except Exception as exc:

session.rollback()

raise exc

finally:

session.close()

tasks.py file code is

import asyncio

import logging

from asyncio import ensure\_future

from functools import wraps

from traceback import format\_exception

from typing import Any, Callable, Coroutine, Optional, Union

from starlette.concurrency import run\_in\_threadpool

NoArgsNoReturnFuncT = Callable[[], None]

NoArgsNoReturnAsyncFuncT = Callable[[], Coroutine[Any, Any, None]]

NoArgsNoReturnDecorator = Callable[[Union[NoArgsNoReturnFuncT, NoArgsNoReturnAsyncFuncT]], NoArgsNoReturnAsyncFuncT]

def repeat\_every(

\*,

seconds: float,

wait\_first: bool = False,

logger: Optional[logging.Logger] = None,

raise\_exceptions: bool = False,

max\_repetitions: Optional[int] = None,

) -> NoArgsNoReturnDecorator:

def decorator(func: Union[NoArgsNoReturnAsyncFuncT, NoArgsNoReturnFuncT]) -> NoArgsNoReturnAsyncFuncT:

is\_coroutine = asyncio.iscoroutinefunction(func)

@wraps(func)

async def wrapped() -> None:

repetitions = 0

async def loop() -> None:

nonlocal repetitions

if wait\_first:

await asyncio.sleep(seconds)

while max\_repetitions is None or repetitions < max\_repetitions:

try:

if is\_coroutine:

await func() # type: ignore

else:

await run\_in\_threadpool(func)

repetitions += 1

except Exception as exc:

if logger is not None:

formatted\_exception = "".join(format\_exception(type(exc), exc, exc.\_\_traceback\_\_))

logger.error(formatted\_exception)

if raise\_exceptions:

raise exc

await asyncio.sleep(seconds)

ensure\_future(loop())

return wrapped

return decorator

timing.py file code is

"""

Based on https://github.com/steinnes/timing-asgi.git

"""

import resource

import time

from typing import Any, Callable, Optional

from fastapi import FastAPI

from starlette.middleware.base import RequestResponseEndpoint

from starlette.requests import Request

from starlette.responses import Response

from starlette.routing import Match, Mount

class TimingStats:

def **init**(

self, name: Optional[str] = None, record: Callable[[str], None] = None, exclude: Optional[str] = None

) -> None:

self.name = name

self.record = record or print

self.start\_time: float = 0

self.start\_cpu\_time: float = 0

self.end\_cpu\_time: float = 0

self.end\_time: float = 0

self.silent: bool = False

if self.name and exclude and (exclude in self.name):

self.silent = True

def start(self) -> None:

self.start\_time = time.time()

self.start\_cpu\_time = get\_cpu\_time()

def take\_split(self) -> None:

self.end\_time = time.time()

self.end\_cpu\_time = get\_cpu\_time()

@property

def time(self) -> float:

return self.end\_time - self.start\_time

@property

def cpu\_time(self) -> float:

return self.end\_cpu\_time - self.start\_cpu\_time

def **enter**(self) -> "TimingStats":

self.start()

return self

def **exit**(self, exc\_type: Any, exc\_value: Any, traceback: Any) -> None:

self.emit()

def emit(self, note: Optional[str] = None) -> None:

if not self.silent:

self.take\_split()

cpu\_ms = 1000 \* self.cpu\_time

wall\_ms = 1000 \* self.time

message = f"TIMING: Wall: {wall\_ms:6.1f}ms | CPU: {cpu\_ms:6.1f}ms | {self.name}"

if note is not None:

message += f" ({note})"

self.record(message)

class MetricNamer:

def **init**(self, prefix: str, app: FastAPI):

if prefix:

prefix += "."

self.prefix = prefix

self.app = app

def **call**(self, scope: Any) -> str:

route = None

for r in self.app.router.routes:

if r.matches(scope)[0] == Match.FULL:

route = r

break

if hasattr(route, "endpoint") and hasattr(route, "name"):

name = f"{self.prefix}{route.endpoint.\_\_module\_\_}.{route.name}" # type: ignore

elif isinstance(route, Mount):

name = f"{type(route.app).\_\_name\_\_}<{route.name!r}>"

else:

name = str(f"<Path: {scope['path']}>")

return name

def get\_cpu\_time() -> float:

# taken from timing-asgi

resources = resource.getrusage(resource.RUSAGE\_SELF)

# add up user time (ru\_utime) and system time (ru\_stime)

return resources[0] + resources[1]

def add\_timing\_middleware(

app: FastAPI, record: Callable[[str], None] = None, prefix: str = "", exclude: Optional[str] = None

) -> None:

"""

Don't print timings if exclude occurs as an exact substring of the generated metric name

"""

metric\_namer = MetricNamer(prefix=prefix, app=app)

@app.middleware("http")

async def timing\_middleware(request: Request, call\_next: RequestResponseEndpoint) -> Response:

metric\_name = metric\_namer(request.scope)

with TimingStats(metric\_name, record=record, exclude=exclude) as timer:

request.state.timer = timer

response = await call\_next(request)

return response

def record\_timing(request: Request, note: Optional[str] = None) -> None:

"""

Call this function anywhere you want to display performance information while handling a single request

"""

if hasattr(request.state, "timer"):

assert isinstance(request.state.timer, TimingStats)

request.state.timer.emit(note)

else:

print("TIMING ERROR: No timer present on request")

and last one README file inside the repository is

FastAPI Auth

**Pluggable auth for use with FastAPI**

* Supports OAuth2 Password Flow
* Uses JWT access and refresh tokens
* 100% mypy and test coverage
* Supports custom user models (both ORM and pydantic) without sacrificing any type-safety

Usage:

After installing the development dependencies, the following script should run as-is:

from typing import Optional

import sqlalchemy as sa

from fastapi import FastAPI

from pydantic import EmailStr

from fastapi\_auth.auth\_app import BaseAuthRouterBuilder

from fastapi\_auth.auth\_settings import get\_auth\_settings

from fastapi\_auth.fastapi\_util.api\_model import APIModel

from fastapi\_auth.fastapi\_util.orm.base import Base

from fastapi\_auth.models.user import (

UserBaseInDB as BaseUserModel,

UserCreate as BaseUserCreate,

UserCreateRequest as BaseUserCreateRequest,

UserInDB as BaseUserInDB,

UserUpdate as BaseUserUpdate,

)

from fastapi\_auth.orm.user import BaseUser

# Pydantic Models

class ExtraUserAttributes(APIModel):

email: Optional[EmailStr]

class UserCreate(BaseUserCreate, ExtraUserAttributes):

pass

class UserCreateRequest(BaseUserCreateRequest, ExtraUserAttributes):

pass

class UserInDB(BaseUserInDB, ExtraUserAttributes):

pass

class UserUpdate(BaseUserUpdate, ExtraUserAttributes):

pass

class UserResult(BaseUserModel, ExtraUserAttributes):

pass

# Sqlalchemy Model

class User(BaseUser, Base):

email = sa.Column(sa.String)

class AuthRouterBuilder(

BaseAuthRouterBuilder[

UserCreate, UserCreateRequest, UserInDB, UserUpdate, UserResult, User

]

):

create\_type = UserCreate

create\_request\_type = UserCreateRequest

in\_db\_type = UserInDB

update\_type = UserUpdate

api\_type = UserResult

orm\_type = User

auth\_settings = get\_auth\_settings()

router\_builder = AuthRouterBuilder(auth\_settings)

app = FastAPI()

... # Add routes

router\_builder.include\_auth(app.router)

router\_builder.add\_expired\_token\_cleanup(app)

print(list(app.openapi()["paths"].keys()))

"""

[

"/auth/token",

"/auth/token/refresh",

"/auth/token/validate",

"/auth/token/logout",

"/auth/token/logout/all",

"/auth/register",

"/auth/self",

"/admin/users/{user\_id}",

"/admin/users",

]

"""

You can run the above app the same way you would run any other ASGI app, and see the docs at /docs.

* You can find a more complete example of configuring an app in tests/test\_auth\_app/build\_app.py.
* Dependency functions that can be used to read the user can be found in fastapi\_auth.dependencies
  + If you want to inject the full user model from the database, use the classmethod AuthRouteBuilder.get\_user
* Various environment-variable-controlled settings are contained in fastapi\_auth.auth\_settings

all the information is present in that GitHub repository and the the name of that repository is fastapi-auth.

I have also provided one more repository link called fastapi-sqlalchemy.

I am attaching a file that help you understand all the information inside their repository.

read that information carefully.

inside the git repository there are fast-auth folderss. inside that folder there are features, orm, setting, setup and util folder are present and also api\_model.py file. code inside the api\_model.py file from functools import partial from pydantic import BaseConfig, BaseModel from fastapi\_auth.fastapi\_util.util.camelcase import snake2camel class APIModel(BaseModel): class Config(BaseConfig): orm\_mode = True allow\_population\_by\_field\_name = True alias\_generator = partial(snake2camel, start\_lower=True) class APIMessage(APIModel): detail: str inside the features folder, there are program files. cbv.py and inferring\_routher.py files code. code inside cbv.py file. import inspect from typing import Any, Callable, List, Type, TypeVar, Union, get\_type\_hints from fastapi import APIRouter, Depends from pydantic.typing import is\_classvar from starlette.routing import Route, WebSocketRoute T = TypeVar("T") CBV\_CLASS\_KEY = "\_\_cbv\_class\_\_" def cbv(router: APIRouter) -> Callable[[Type[T]], Type[T]]: def decorator(cls: Type[T]) -> Type[T]: return \_cbv(router, cls) return decorator def \_cbv(router: APIRouter, cls: Type[T]) -> Type[T]: \*init\*cbv(cls) cbv\_router = APIRouter() functions = inspect.getmembers(cls, inspect.isfunction) routes\_by\_endpoint = { route.endpoint: route for route in router.routes if isinstance(route, (Route, WebSocketRoute)) } for \_, func in functions: route = routes\_by\_endpoint.get(func) if route is None: continue router.routes.remove(route) \*update\*cbv\_route\_endpoint\_signature(cls, route) cbv\_router.routes.append(route) router.include\_router(cbv\_router) return cls def \*update\*cbv\_route\_endpoint\_signature(cls: Type[Any], route: Union[Route, WebSocketRoute]) -> None: old\_endpoint = route.endpoint old\_signature = inspect.signature(old\_endpoint) old\_parameters: List[inspect.Parameter] = list(old\_signature.parameters.values()) old\_first\_parameter = old\_parameters[0] new\_first\_parameter = old\_first\_parameter.replace(default=Depends(cls)) new\_parameters = [new\_first\_parameter] + [ parameter.replace(kind=inspect.Parameter.KEYWORD\_ONLY) for parameter in old\_parameters[1:] ] new\_signature = old\_signature.replace(parameters=new\_parameters) setattr(route.endpoint, "\_\_signature\_\_", new\_signature) def \*init\*cbv(cls: Type[Any]) -> None: if getattr(cls, CBV\_CLASS\_KEY, False): # pragma: no cover return # Already initialized old\_init: Callable[..., Any] = cls.\_\_init\_\_ old\_signature = inspect.signature(old\_init) old\_parameters = list(old\_signature.parameters.values())[1:] # drop `self` parameter new\_parameters = [ x for x in old\_parameters if x.kind not in (inspect.Parameter.VAR\_POSITIONAL, inspect.Parameter.VAR\_KEYWORD) ] dependency\_names: List[str] = [] for name, hint in get\_type\_hints(cls).items(): if is\_classvar(hint): continue parameter\_kwargs = {} parameter\_kwargs["default"] = getattr(cls, name, Ellipsis) dependency\_names.append(name) new\_parameters.append( inspect.Parameter(name=name, kind=inspect.Parameter.KEYWORD\_ONLY, annotation=hint, \*\*parameter\_kwargs) ) new\_signature = old\_signature.replace(parameters=new\_parameters) def new\_init(self: Any, \*args: Any, \*\*kwargs: Any) -> None: for dep\_name in dependency\_names: dep\_value = kwargs.pop(dep\_name) setattr(self, dep\_name, dep\_value) old\_init(self, \*args, \*\*kwargs) setattr(cls, "\_\_signature\_\_", new\_signature) setattr(cls, "\_\_init\_\_", new\_init) setattr(cls, CBV\_CLASS\_KEY, True) code inside inferring\_router.py. from typing import TYPE\_CHECKING, Any, Callable, get\_type\_hints from fastapi import APIRouter class InferringRouter(APIRouter): if not TYPE\_CHECKING: def add\_api\_route(self, path: str, endpoint: Callable[..., Any], \*\*kwargs: Any) -> None: if kwargs.get("response\_model") is None: kwargs["response\_model"] = get\_type\_hints(endpoint).get("return") return super().add\_api\_route(path, endpoint, \*\*kwargs) else: # pragma: no cover pass orm folder and the inside files base.py, columns.py and guid\_type.py base.py file and there code. from typing import TYPE\_CHECKING, Any, Dict, TypeVar import sqlalchemy as sa from sqlalchemy.ext.declarative import DeclarativeMeta, declarative\_base, declared\_attr from sqlalchemy.orm import Session from sqlalchemy.sql.base import ImmutableColumnCollection from fastapi\_auth.fastapi\_util.util.camelcase import camel2snake T = TypeVar("T", bound="CustomBase") MYPY = False class CustomMeta(DeclarativeMeta): \*\*table\*\*: sa.Table @property def columns(cls) -> ImmutableColumnCollection: return cls.\_\_table\_\_.columns class CustomBase: \*\*table\*\*: sa.Table if TYPE\_CHECKING: \*\*tablename\*\*: str else: @declared\_attr def \*\*tablename\*\*(cls) -> str: return camel2snake(cls.\_\_name\_\_) def dict(self) -> Dict[str, Any]: data = {key: getattr(self, key) for key in self.\_\_table\_\_.c.keys()} return data \*Base = declarative\*base(cls=CustomBase, metaclass=CustomMeta) if TYPE\_CHECKING: # This is necessary for pycharm, but not mypy class Base(\_Base, CustomBase, metaclass=CustomMeta): \*\*table\*\*: sa.Table \_\_tablename\_: str metadata: sa.MetaData columns: ImmutableColumnCollection if not MYPY: # pragma: no cover # Suppress pycharm kwargs warnings def \*\*init\*\*(self, \*\*kwargs: Any) -> None: pass def dict(self) -> Dict[str, Any]: ... else: exec("Base = \_Base") # Need to hide from PyCharm S = TypeVar("S", bound="Base") def add\_base(session: Session, item: S) -> S: session.add(item) session.commit() return item code inside columns.py file. from datetime import datetime from typing import Any, Dict, Optional, Type, TypeVar, Union, overload from uuid import UUID, uuid4 import sqlalchemy as sa from sqlalchemy.dialects.postgresql import JSONB from typing\_extensions import Literal from fastapi\_auth.fastapi\_util.orm.guid\_type import GUID from fastapi\_auth.fastapi\_util.settings.database\_settings import DatabaseBackend, get\_database\_settings IdentifierT = TypeVar("IdentifierT", bound=UUID) @overload def pk\_column(id\_type: Type[IdentifierT]) -> "sa.Column[IdentifierT]": ... @overload def pk\_column(id\_type: None = None) -> "sa.Column[UUID]": ... def pk\_column(id\_type: Optional[Type[IdentifierT]] = None) -> "Union[sa.Column[IdentifierT], sa.Column[UUID]]": """ The server-default value should be updated in the metadata later """ using\_postgres = get\_database\_settings().backend == DatabaseBackend.postgresql default\_kwargs: Dict[str, Any] = {"default": uuid4} if not using\_postgres else { "server\_default": sa.text("gen\_random\_uuid()") } return sa.Column(GUID, primary\_key=True, index=True, \*\*default\_kwargs) @overload def fk\_column( column: Union[str, "sa.Column[IdentifierT]"], nullable: Literal[True], index: bool = False, primary\_key: bool = False, unique: bool = False, ) -> "sa.Column[Optional[IdentifierT]]": ... @overload def fk\_column( column: Union[str, "sa.Column[IdentifierT]"], nullable: Literal[False] = False, index: bool = False, primary\_key: bool = False, unique: bool = False, ) -> "sa.Column[IdentifierT]": ... def fk\_column( column: Union[str, "sa.Column[IdentifierT]"], nullable: bool = False, index: bool = False, primary\_key: bool = False, unique: bool = False, ) -> "Union[sa.Column[IdentifierT], sa.Column[Optional[IdentifierT]]]": return sa.Column( # type: ignore GUID, sa.ForeignKey(column, ondelete="CASCADE"), index=index, nullable=nullable, primary\_key=primary\_key, unique=unique, ) def json\_column(\*, nullable: bool) -> "sa.Column[Dict[str, Any]]": using\_postgres = get\_database\_settings().backend == DatabaseBackend.postgresql column\_type = JSONB() if using\_postgres else sa.JSON() return sa.Column(column\_type, nullable=nullable) # type: ignore def created\_at\_column() -> "sa.Column[datetime]": return sa.Column(sa.DateTime(timezone=True), server\_default=sa.func.now(), nullable=False) def updated\_at\_column() -> "sa.Column[datetime]": return sa.Column(sa.DateTime(timezone=True), server\_default=sa.func.now(), onupdate=sa.func.now(), nullable=False) guid\_type.py file and there inside code. import uuid from typing import no\_type\_check from sqlalchemy.dialects.postgresql.base import UUID from sqlalchemy.sql.sqltypes import CHAR from sqlalchemy.sql.type\_api import TypeDecorator class GUID(TypeDecorator): # type: ignore """ Platform-independent GUID type. Uses PostgreSQL's UUID type, otherwise uses CHAR(32), storing as stringified hex values. Taken from SQLAlchemy docs: https://docs.sqlalchemy.org/en/13/core/custom\_types.html#backend-agnostic-guid-type """ impl = CHAR @no\_type\_check def \*\*init\*\*(self, \*args, \*\*kwargs): super().\_\_init\_\_(\*args, \*\*kwargs) @no\_type\_check def load\_dialect\_impl(self, dialect): if dialect.name == "postgresql": return dialect.type\_descriptor(UUID()) else: return dialect.type\_descriptor(CHAR(32)) @no\_type\_check def process\_bind\_param(self, value, dialect): if value is None: return value elif dialect.name == "postgresql": return str(value) else: if not isinstance(value, uuid.UUID): return "%.32x" % uuid.UUID(value).int else: # hexstring return "%.32x" % value.int @no\_type\_check def process\_result\_value(self, value, dialect): if value is None: return value else: if not isinstance(value, uuid.UUID): value = uuid.UUID(value) return value inside the setting folder ther are also programing file known as api\_settings.py, base\_api\_settings.py, and database\_setting.py files. code inside the api\_settings.py file is from functools import lru\_cache from typing import Any, Dict from fastapi\_auth.fastapi\_util.settings.base\_api\_settings import BaseAPISettings class APISettings(BaseAPISettings): # fastapi.applications.FastAPI initializer kwargs debug: bool = False docs\_url: str = "/docs" openapi\_prefix: str = "" openapi\_url: str = "/openapi.json" redoc\_url: str = "/redoc" title: str = "Fast API" version: str = "0.1.0" # Custom settings disable\_docs: bool = False disable\_superuser\_dependency: bool = False include\_admin\_routes: bool = False main\_router\_prefix: str = "/api/v1" @property def fastapi\_kwargs(self) -> Dict[str, Any]: fastapi\_kwargs: Dict[str, Any] = { "debug": self.debug, "docs\_url": self.docs\_url, "openapi\_prefix": self.openapi\_prefix, "openapi\_url": self.openapi\_url, "redoc\_url": self.redoc\_url, "title": self.title, "version": self.version, } if self.disable\_docs: fastapi\_kwargs.update({"docs\_url": None, "openapi\_url": None, "redoc\_url": None}) return fastapi\_kwargs class Config: env\_prefix = "api\_" @lru\_cache() def get\_api\_settings() -> APISettings: return APISettings() code inside base\_api\_settings.py file. from pydantic import BaseSettings class BaseAPISettings(BaseSettings): class Config: env\_prefix = "" arbitrary\_types\_allowed = True validate\_assignment = True database\_setting.py file code is from enum import auto from functools import lru\_cache from typing import Any, Dict, Optional import sqlalchemy as sa from pydantic import validator from fastapi\_auth.fastapi\_util.settings.base\_api\_settings import BaseAPISettings from fastapi\_auth.fastapi\_util.util.enums import StrEnum class DatabaseBackend(StrEnum): postgresql = auto() sqlite = auto() @staticmethod def from\_engine(engine: sa.engine.Engine) -> "DatabaseBackend": return DatabaseBackend(engine.dialect.name) class DatabaseSettings(BaseAPISettings): backend: DatabaseBackend = None # type: ignore user: Optional[str] password: Optional[str] host: Optional[str] db: Optional[str] sqlalchemy\_uri: str = None # type: ignore log\_sqlalchemy\_sql\_statements: bool = False min\_size: int = 10 max\_size: int = 10 force\_rollback: bool = False @validator("sqlalchemy\_uri", pre=True, always=True) def validate\_sqlalchemy\_uri(cls, v: Optional[str], values: Dict[str, Any]) -> str: if v is None: backend = values.get("backend") backend = backend.value if backend is not None else None user = values["user"] password = values["password"] host = values["host"] db = values["db"] v = f"{backend}://{user}:{password}@{host}/{db}" return v class Config: env\_prefix = "db\_" @lru\_cache() def get\_database\_settings() -> DatabaseSettings: return DatabaseSettings() setup folder and other files are initialize.py, setup\_api.py, and setup\_database.py initialize.py file and there code import sqlalchemy as sa from fastapi import FastAPI from fastapi\_auth.fastapi\_util.orm.base import Base from fastapi\_auth.fastapi\_util.setup.setup\_database import setup\_database, setup\_database\_metadata from fastapi\_auth.fastapi\_util.util.session import get\_engine def initialize\_database(engine: sa.engine.Engine) -> None: setup\_database(engine) def get\_configured\_metadata(\_app: FastAPI) -> sa.MetaData: """ This function accepts the app instance as an argument purely as a check to ensure that all resources the app depends on have been imported. In particular, this ensures the sqlalchemy metadata is populated. """ engine = get\_engine() setup\_database(engine) setup\_database\_metadata(Base.metadata, engine) return Base.metadata setup\_api.py file and other code is from fastapi import FastAPI from fastapi.routing import APIRoute def setup\_openapi(app: FastAPI) -> None: """ Simplify operation IDs so that generated clients have simpler api function names """ for route in app.routes: if isinstance(route, APIRoute): route.operation\_id = route.name setup\_database.py file and other code is import sqlalchemy as sa from fastapi\_auth.fastapi\_util.orm.guid\_type import GUID from fastapi\_auth.fastapi\_util.settings.database\_settings import DatabaseBackend def setup\_database(engine: sa.engine.Engine) -> None: setup\_guids(engine) def setup\_guids(engine: sa.engine.Engine) -> None: """ Set up UUID generation using the uuid-ossp extension for postgres """ database\_backend = DatabaseBackend.from\_engine(engine) # TODO: Add some way to run postgres-specific tests if database\_backend == DatabaseBackend.postgresql: # pragma: no cover # noinspection SqlDialectInspection,SqlNoDataSourceInspection uuid\_generation\_setup\_query = 'create EXTENSION if not EXISTS "pgcrypto"' engine.execute(uuid\_generation\_setup\_query) def setup\_database\_metadata(metadata: sa.MetaData, engine: sa.engine.Engine) -> None: setup\_guid\_server\_defaults(metadata, engine) def setup\_guid\_server\_defaults(metadata: sa.MetaData, engine: sa.engine.Engine) -> None: database\_backend = DatabaseBackend.from\_engine(engine) guid\_server\_defaults = { DatabaseBackend.postgresql: "gen\_random\_uuid()", DatabaseBackend.sqlite: "(lower(hex(randomblob(16))))", } for table in metadata.tables.values(): if len(table.primary\_key.columns) != 1: continue for column in table.primary\_key.columns: if type(column.type) is GUID: column.server\_default = sa.DefaultClause(sa.text(guid\_server\_defaults[database\_backend])) util folder and other files are camelcase.py, enums.py, session.py, tasks.py, and timing.py code inside camelcase.py file import re def snake2camel(snake: str, start\_lower: bool = False) -> str: camel = snake.title() camel = re.sub("([0-9A-Za-z])\_(?=[0-9A-Z])", lambda m: m.group(1), camel) if start\_lower: camel = re.sub("(^\_\*[A-Z])", lambda m: m.group(1).lower(), camel) return camel def camel2snake(camel: str) -> str: snake = re.sub(r"([a-zA-Z])([0-9])", lambda m: f"{m.group(1)}\_{m.group(2)}", camel) snake = re.sub(r"([a-z0-9])([A-Z])", lambda m: f"{m.group(1)}\_{m.group(2)}", snake) return snake.lower() enums.py file code is from enum import Enum from fastapi\_auth.fastapi\_util.util.camelcase import snake2camel class StrEnum(str, Enum): # noinspection PyMethodParameters def \*generate\*next\_value\_(name, start, count, last\_values) -> str: # type: ignore """ Uses the name as the automatic value, rather than an integer See https://docs.python.org/3/library/enum.html#using-automatic-values for reference """ return name class CamelStrEnum(str, Enum): # noinspection PyMethodParameters def \*generate\*next\_value\_(name, start, count, last\_values) -> str: # type: ignore """ Uses the camelCase name as the automatic value, rather than an integer See https://docs.python.org/3/library/enum.html#using-automatic-values for reference """ return snake2camel(name, start\_lower=True) session.py file code is import logging from contextlib import contextmanager from functools import lru\_cache from typing import Iterator, Optional import sqlalchemy as sa from sqlalchemy.orm import Session from fastapi\_auth.fastapi\_util.settings.database\_settings import DatabaseBackend, get\_database\_settings @lru\_cache() def get\_engine() -> sa.engine.Engine: db\_settings = get\_database\_settings() uri = db\_settings.sqlalchemy\_uri log\_sqlalchemy\_sql\_statements = db\_settings.log\_sqlalchemy\_sql\_statements database\_backend = db\_settings.backend return get\_new\_engine(uri, log\_sqlalchemy\_sql\_statements, database\_backend) @lru\_cache() def get\_sessionmaker() -> sa.orm.sessionmaker: return get\_sessionmaker\_for\_engine(get\_engine()) def get\_new\_engine( uri: str, log\_sqlalchemy\_sql\_statements: bool = False, database\_backend: DatabaseBackend = DatabaseBackend.postgresql, ) -> sa.engine.Engine: if log\_sqlalchemy\_sql\_statements: logging.getLogger("sqlalchemy.engine").setLevel(logging.INFO) else: logging.getLogger("sqlalchemy.engine").setLevel(logging.ERROR) kwargs = {} if database\_backend == DatabaseBackend.sqlite: kwargs.update({"connect\_args": {"check\_same\_thread": False}}) return sa.create\_engine(uri, pool\_pre\_ping=True, \*\*kwargs) def get\_sessionmaker\_for\_engine(engine: sa.engine.Engine) -> sa.orm.sessionmaker: return sa.orm.sessionmaker(autocommit=False, autoflush=False, bind=engine) def get\_session() -> sa.orm.Session: return get\_sessionmaker()() @contextmanager def context\_session(engine: Optional[sa.engine.Engine] = None) -> Iterator[Session]: yield from \*get\*db(engine) def get\_db() -> Iterator[Session]: """ Intended for use as a FastAPI dependency """ yield from \*get\*db() def \*get\*db(engine: Optional[sa.engine.Engine] = None) -> Iterator[Session]: if engine is None: session = get\_session() else: session = get\_sessionmaker\_for\_engine(engine)() try: yield session session.commit() except Exception as exc: session.rollback() raise exc finally: session.close() tasks.py file code is import asyncio import logging from asyncio import ensure\_future from functools import wraps from traceback import format\_exception from typing import Any, Callable, Coroutine, Optional, Union from starlette.concurrency import run\_in\_threadpool NoArgsNoReturnFuncT = Callable[[], None] NoArgsNoReturnAsyncFuncT = Callable[[], Coroutine[Any, Any, None]] NoArgsNoReturnDecorator = Callable[[Union[NoArgsNoReturnFuncT, NoArgsNoReturnAsyncFuncT]], NoArgsNoReturnAsyncFuncT] def repeat\_every( \*, seconds: float, wait\_first: bool = False, logger: Optional[logging.Logger] = None, raise\_exceptions: bool = False, max\_repetitions: Optional[int] = None, ) -> NoArgsNoReturnDecorator: def decorator(func: Union[NoArgsNoReturnAsyncFuncT, NoArgsNoReturnFuncT]) -> NoArgsNoReturnAsyncFuncT: is\_coroutine = asyncio.iscoroutinefunction(func) @wraps(func) async def wrapped() -> None: repetitions = 0 async def loop() -> None: nonlocal repetitions if wait\_first: await asyncio.sleep(seconds) while max\_repetitions is None or repetitions < max\_repetitions: try: if is\_coroutine: await func() # type: ignore else: await run\_in\_threadpool(func) repetitions += 1 except Exception as exc: if logger is not None: formatted\_exception = "".join(format\_exception(type(exc), exc, exc.\_\_traceback\_\_)) logger.error(formatted\_exception) if raise\_exceptions: raise exc await asyncio.sleep(seconds) ensure\_future(loop()) return wrapped return decorator timing.py file code is """ Based on https://github.com/steinnes/timing-asgi.git """ import resource import time from typing import Any, Callable, Optional from fastapi import FastAPI from starlette.middleware.base import RequestResponseEndpoint from starlette.requests import Request from starlette.responses import Response from starlette.routing import Match, Mount class TimingStats: def \*\*init\*\*( self, name: Optional[str] = None, record: Callable[[str], None] = None, exclude: Optional[str] = None ) -> None: self.name = name self.record = record or print self.start\_time: float = 0 self.start\_cpu\_time: float = 0 self.end\_cpu\_time: float = 0 self.end\_time: float = 0 self.silent: bool = False if self.name and exclude and (exclude in self.name): self.silent = True def start(self) -> None: self.start\_time = time.time() self.start\_cpu\_time = get\_cpu\_time() def take\_split(self) -> None: self.end\_time = time.time() self.end\_cpu\_time = get\_cpu\_time() @property def time(self) -> float: return self.end\_time - self.start\_time @property def cpu\_time(self) -> float: return self.end\_cpu\_time - self.start\_cpu\_time def \*\*enter\*\*(self) -> "TimingStats": self.start() return self def \*\*exit\*\*(self, exc\_type: Any, exc\_value: Any, traceback: Any) -> None: self.emit() def emit(self, note: Optional[str] = None) -> None: if not self.silent: self.take\_split() cpu\_ms = 1000 \* self.cpu\_time wall\_ms = 1000 \* self.time message = f"TIMING: Wall: {wall\_ms:6.1f}ms | CPU: {cpu\_ms:6.1f}ms | {self.name}" if note is not None: message += f" ({note})" self.record(message) class MetricNamer: def \*\*init\*\*(self, prefix: str, app: FastAPI): if prefix: prefix += "." self.prefix = prefix self.app = app def \*\*call\*\*(self, scope: Any) -> str: route = None for r in self.app.router.routes: if r.matches(scope)[0] == Match.FULL: route = r break if hasattr(route, "endpoint") and hasattr(route, "name"): name = f"{self.prefix}{route.endpoint.\_\_module\_\_}.{route.name}" # type: ignore elif isinstance(route, Mount): name = f"{type(route.app).\_\_name\_\_}<{route.name!r}>" else: name = str(f"") return name def get\_cpu\_time() -> float: # taken from timing-asgi resources = resource.getrusage(resource.RUSAGE\_SELF) # add up user time (ru\_utime) and system time (ru\_stime) return resources[0] + resources[1] def add\_timing\_middleware( app: FastAPI, record: Callable[[str], None] = None, prefix: str = "", exclude: Optional[str] = None ) -> None: """ Don't print timings if exclude occurs as an exact substring of the generated metric name """ metric\_namer = MetricNamer(prefix=prefix, app=app) @app.middleware("http") async def timing\_middleware(request: Request, call\_next: RequestResponseEndpoint) -> Response: metric\_name = metric\_namer(request.scope) with TimingStats(metric\_name, record=record, exclude=exclude) as timer: request.state.timer = timer response = await call\_next(request) return response def record\_timing(request: Request, note: Optional[str] = None) -> None: """ Call this function anywhere you want to display performance information while handling a single request """ if hasattr(request.state, "timer"): assert isinstance(request.state.timer, TimingStats) request.state.timer.emit(note) else: print("TIMING ERROR: No timer present on request") and last one README file inside the repository is FastAPI Auth \*\*Pluggable auth for use with FastAPI\*\* \* Supports OAuth2 Password Flow \* Uses JWT access and refresh tokens \* 100% mypy and test coverage \* Supports custom user models (both ORM and pydantic) without sacrificing any type-safety Usage: After installing the development dependencies, the following script should run as-is: ``` from typing import Optional import sqlalchemy as sa from fastapi import FastAPI from pydantic import EmailStr from fastapi\_auth.auth\_app import BaseAuthRouterBuilder from fastapi\_auth.auth\_settings import get\_auth\_settings from fastapi\_auth.fastapi\_util.api\_model import APIModel from fastapi\_auth.fastapi\_util.orm.base import Base from fastapi\_auth.models.user import ( UserBaseInDB as BaseUserModel, UserCreate as BaseUserCreate, UserCreateRequest as BaseUserCreateRequest, UserInDB as BaseUserInDB, UserUpdate as BaseUserUpdate, ) from fastapi\_auth.orm.user import BaseUser # Pydantic Models class ExtraUserAttributes(APIModel): email: Optional[EmailStr] class UserCreate(BaseUserCreate, ExtraUserAttributes): pass class UserCreateRequest(BaseUserCreateRequest, ExtraUserAttributes): pass class UserInDB(BaseUserInDB, ExtraUserAttributes): pass class UserUpdate(BaseUserUpdate, ExtraUserAttributes): pass class UserResult(BaseUserModel, ExtraUserAttributes): pass # Sqlalchemy Model class User(BaseUser, Base): email = sa.Column(sa.String) class AuthRouterBuilder( BaseAuthRouterBuilder[ UserCreate, UserCreateRequest, UserInDB, UserUpdate, UserResult, User ] ): create\_type = UserCreate create\_request\_type = UserCreateRequest in\_db\_type = UserInDB update\_type = UserUpdate api\_type = UserResult orm\_type = User auth\_settings = get\_auth\_settings() router\_builder = AuthRouterBuilder(auth\_settings) app = FastAPI() ... # Add routes router\_builder.include\_auth(app.router) router\_builder.add\_expired\_token\_cleanup(app) print(list(app.openapi()["paths"].keys())) """ [ "/auth/token", "/auth/token/refresh", "/auth/token/validate", "/auth/token/logout", "/auth/token/logout/all", "/auth/register", "/auth/self", "/admin/users/{user\_id}", "/admin/users", ] """ ``` You can run the above app the same way you would run any other ASGI app, and see the docs at `/docs`. \* You can find a more complete example of configuring an app in `tests/test\_auth\_app/build\_app.py`. \* Dependency functions that can be used to read the user can be found in `fastapi\_auth.dependencies` \* If you want to inject the full user model from the database, use the classmethod `AuthRouteBuilder.get\_user` \* Various environment-variable-controlled settings are contained in `fastapi\_auth.auth\_settings` all the information is present in that GitHub repository and the the name of that repository is fastapi-auth. I have also provided one more repository link called fastapi-sqlalchemy. I am attaching a file that help you understand all the information inside their repository. read that information carefully.