Авторизация

Вводная информация

Перед началом создания авторизации необходимо понять, что она будет из себя представлять. При успешной авторизации (Правильном логине и пароле) в БД будет формироваться токен, при помощи которого пользователь будет проходить верификацию. Сам токен это хешированные данные, внутри которого содержатся `id` пользователя и время жизни токена, по истечению которого он является не действительным.

Создание модели токена

В файле user/models.py создадим следующую модель:

```
class Token(Base):
    __tablename__ = "tokens"

    id = Column(Integer, primary_key=True)
    token = Column(String(256), unique=True, nullable=False)
    user_id = Column(Integer, ForeignKey("users.id"))
```

Relationship

Теперь нам необходимо создать связь - `relationship` .
`relationship` — это функция в библиотеке SQLAlchemy, которая используется для установления связей между таблицами баз данных на уровне моделей. Она определяет, как связаны объекты (строки) из одной таблицы с объектами из другой таблицы.

Добавьте в `User` и `Token` соответственно

```
#Это в User
tokens = relationship("Token", back_populates="user")
```

```
#Это в Token
user = relationship("User", back_populates="token")
```

Функции для авторизации

Для работы с токенами нам понядобится определенный функционал, который мы определим внутри моделей `User` и `Token` . Итоговый код представлен ниже

```python
from datetime import datetime, timedelta, timezone
from typing import Optional

from fastapi import HTTPException, status
import jwt
import bcrypt

from sqlalchemy import Column, Integer, String, ForeignKey, Enum
from sqlalchemy.orm import DeclarativeBase, relationship
from sqlalchemy.ext.asyncio import AsyncSession
from enum import Enum as BaseEnum

from config import SECRET_KEY

#Базовые классы и Role

class User(Base):
    __tablename__ = "users"

    id = Column(Integer, primary_key=True)
    username = Column(String(20), unique=True, nullable=False)
    hashed_password = Column(String(512), nullable=False)
    role = Column(Enum(Role), default=Role.USER, nullable=False)
    full_name = Column(String(40), nullable=False)

    tokens = relationship("Token", back_populates="user")

    async def set_password(self, password: str) -> None:
        self.hashed_password = bcrypt.hashpw(
            password.encode("utf-8"), bcrypt.gensalt()
        ).decode("utf-8")

    #Проверка пароля
    async def check_password(self, password: str) -> bool:
        return bcrypt.checkpw(
            password.encode("utf-8"), self.hashed_password.encode("utf-8")
        )

    #Генерация токена
    #expires_in - время жизни токена в сенундах
    # в payload можете увидеть подробнее что содержет
    async def generate_token(self, expires_in: int = 4800) -> str:
        payload = {
```

```
            "user_id": self.id,
            "exp": datetime.now(timezone.utc) + timedelta(seconds=expires_in),
        }
        return jwt.encode(payload, SECRET_KEY, algorithm="HS256")


class Token(Base):
    __tablename__ = "tokens"

    id = Column(Integer, primary_key=True)
    token = Column(String(256), unique=True, nullable=False)
    user_id = Column(Integer, ForeignKey("users.id"))

    user = relationship("User", back_populates="token")

    #Верификация токена, если действителен, то возвращает токен и сообщение
    #Инче отправляет на обновление
    #Если пользователь не авторизовант возвращет HTTP_401_UNAUTHORIZED
    async def verify_token(self, session: AsyncSession, user: Optional[User]):
        try:
            jwt.decode(self.token, SECRET_KEY, algorithms=["HS256"])
            return status.HTTP_200_OK, "Токен верефицирован", self

        except jwt.ExpiredSignatureError:
            return await self.refresh_token(session, user)

        except jwt.InvalidTokenError:
            raise HTTPException(
                status_code=status.HTTP_401_UNAUTHORIZED,
                detail="Неправильный токен",
                headers={"WWW-Authenticate": "Bearer"},
            )

    #Обновление токена
    async def refresh_token(self, session: AsyncSession, user: Optional[User]):
        if user is None:
            raise HTTPException(
                status_code=status.HTTP_401_UNAUTHORIZED,
                detail="Токен истёк",
                headers={"WWW-Authenticate": "Bearer"},
            )

        new_token = await user.generate_token()
```

```
        self.token = new_token
        session.add(self)
        await session.commit()

        return status.HTTP_200_OK, "Токен обновлён", self
```

Auth Query

В файле `auth.queries.py` создадим функцию функции `get_user_token` и `login`

```python
#Добавьте импорты
from fastapi import Depends, HTTPException, status
from fastapi.security import OAuth2PasswordBearer
from sqlalchemy import select
from sqlalchemy.ext.asyncio import AsyncSession

from ..database import get_session
from ..user.queries import get_user_by_username
from ..user.models import User, Token
from ..user.schemes import UserCreate, BaseUser

#Добавьте функции
async def get_user_token(session: AsyncSession, user_id: int):
    result = await session.execute(select(Token).where(Token.user_id == user_id))

    return result.scalar_one_or_none()


async def login(session: AsyncSession, login: str, password: str) -> Token:

    user = await get_user_by_username(session, login)

    if user is None:
        raise HTTPException(status.HTTP_404_NOT_FOUND, "Пользователь не найден")

    correct_password = await user.check_password(password)

    if not correct_password:
        raise HTTPException(status.HTTP_401_UNAUTHORIZED, detail="Неверный пароль")

    token = await get_user_token(session, user.id)

    if token is None:
        token = await user.generate_token()
        token = Token(user_id=user.id, token=token)
        status_code = status.HTTP_201_CREATED
        msg = "Пользователь зарегестрирован"

        session.add(token)
        await session.commit()

    else:
        status_code, msg, token = await token.verify_token(session, user)
```

```
    return status_code, msg, token.token
```

Auth Endpoint

Теперь можем приступить к созданию ендпоинта авторизации, для нам понадобится LoginForm этого в файл `auth.router` добавим следующий код

Схема

```
class LoginForm(BaseModel):
    username: str
    password: str
    client_id: Optional[int] = None
    client_secret: Optional[str] = None

    @classmethod
    def as_form(
        cls,
        username: str = Form(...),
        password: str = Form(...),
        client_id: Optional[int] = Form(None),
        client_secret: Optional[str] = Form(None),
    ):
        return cls(
            username=username,
            password=password,
            client_id=client_id,
            client_secret=client_secret,
        )
```

Метод as_form нужен, чтобы кастомная авторизация грамотно работала со Swagger

Ендпоинт

```python
#Импорты
from fastapi import Depends, APIRouter, status
from fastapi.responses import JSONResponse
from sqlalchemy.ext.asyncio import AsyncSession

from ..user.schemes import UserCreate, UserResponse
from ..database import get_session

from .schemes import LoginForm, TokenResponse
from . import queries as qr
from .validators import RegistrationValidator

#Ендпоинт
@router.post("/login", response_class=JSONResponse)
async def get_token(
    login_form: LoginForm = Depends(LoginForm.as_form),
    session: AsyncSession = Depends(get_session),
):
    code, message, token = await qr.login(
        session, login_form.username, login_form.password
    )
    return JSONResponse(
        content=TokenResponse(message=message, access_token=token).dict(),
        status_code=code,
    )
```

Запустите проект и проверьте результат