FastAPI: Users | Я.Шпора

Для управления пользователями можно применить популярную библиотеку FastAPI Users:

```
pip install "fastapi-users[sqlalchemy]==13.0.0"
```

Для работы с библиотекой FastAPI Users необходимы:

1. **Pydantic-схемы пользователя**

```
from fastapi_users import schemas


class UserRead(schemas.BaseUser[int]):
    pass


class UserCreate(schemas.BaseUserCreate):
    pass


class UserUpdate(schemas.BaseUserUpdate):
    pass
```

- `schemas.BaseUser` — схема с базовыми полями модели пользователя: `id`, `email`, `password`, `is_active`, `is_superuser`, `is_verified`. В квадратных скобках для аннотирования указывается тип данных для id пользователя, в примере это `int (Integer)` — целое число.
- `schemas.BaseUserCreate` — схема для создания пользователя; в неё обязательно должны быть переданы `email` и `password`. Любые другие поля, передаваемые в запросе на создание пользователя, будут проигнорированы.
- `schemas.BaseUserUpdate` — схема для обновления объекта пользователя; содержит все базовые поля модели пользователя (в том числе и пароль). Все поля опциональны. Если запрос передаёт обычный пользователь (а не суперпользователь), то поля `is_active`, `is_superuser`, `is_verified` исключаются из набора данных: эти три поля может изменить только суперпользователь.

2. **Модель пользователя**

```
from fastapi_users_db_sqlalchemy import SQLAlchemyBaseUserTable

from app.core.db import Base


class User(SQLAlchemyBaseUserTable[int], Base):
    pass
```

Токены для пользователей генерируются на основе строки, которая задаётся разработчиком в .env-файле. Необходимо добавить эту строку в класс настроек приложения:

```python
from pydantic_settings import BaseSettings, SettingsConfigDict


class Settings(BaseSettings):
    ...
    secret: str = 'SECRET'

    model_config = SettingsConfigDict(env_file='.env')


settings = Settings()
```

Конфигурация библиотеки FastAPI Users

Конфигурацию обычно описывают в файле *app/core/user.py*. Она состоит из нескольких частей.

1. Необходимые импорты и асинхронный генератор доступа к БД:

```python
# Файл app/core/user.py

from typing import Annotated

from fastapi import Depends, Request
from fastapi_users import (
    BaseUserManager, FastAPIUsers, IntegerIDMixin, InvalidPasswordException
```

```
)
from fastapi_users.authentication import (
    AuthenticationBackend, BearerTransport, JWTStrategy
)
from fastapi_users_db_sqlalchemy import SQLAlchemyUserDatabase
from sqlalchemy.ext.asyncio import AsyncSession

from app.core.config import settings
from app.core.db import get_async_session
from app.models.user import User
from app.schemas.user import UserCreate


async def get_user_db(
    session: Annotated[AsyncSession, Depends(get_async_session)]
):
    yield SQLAlchemyUserDatabase(session, User)
```

2. Компоненты, необходимые для построения аутентификационного бэкенда — транспорт, стратегия и сам объект бэкенда.

```
# Файл app/core/user.py

# Определяем транспорт для токена Authorization: Bearer.
# Указываем URL эндпоинта для получения токена.
```

```python
bearer_transport = BearerTransport(tokenUrl='auth/jwt/login')

# Определяем стратегию: хранение токена в виде JWT.
def get_jwt_strategy() -> JWTStrategy:
    # В специальный класс из настроек приложения
    # передаётся секретное слово, используемое для генерации токена.
    # Вторым аргументом передаём срок действия токена в секундах.
    return JWTStrategy(secret=settings.secret, lifetime_seconds=3600)

# Создаём объект бэкенда аутентификации с выбранными параметрами.
auth_backend = AuthenticationBackend(
    name='jwt',  # Произвольное имя бэкенда (должно быть уникальным).
    transport=bearer_transport,
    get_strategy=get_jwt_strategy,
)
```

3. Класс `UserManager` и корутина, возвращающая объект этого класса.

```python
# Файл app/core/user.py

class UserManager(IntegerIDMixin, BaseUserManager[User, int]):

    # Здесь можно описать свои условия валидации пароля.
    # При успешной валидации функция ничего не возвращает.
    # При ошибке валидации будет вызван специальный класс ошибки
```

```
    # InvalidPasswordException.
    async def validate_password(
        self,
        password: str,
        user: UserCreate | User,
    ) -> None:
        if len(password) < 3:
            error = 'Пароль должен содержать не менее 3 символов'
            raise InvalidPasswordException(
                reason=error
            )
        if user.email in password:
            error = 'Пароль не может содержать ваш email'
            raise InvalidPasswordException(
                reason=error
            )

    # Пример метода для действий после успешной регистрации пользователя.
    async def on_after_register(
            self,
            user: User,
            request: Request | None = None,
    ):
        # Вместо print здесь можно настроить отправку письма
        # или переадресацию пользователя на определённую страницу.
```

```
        print(f'Пользователь {user.email} зарегистрирован.')

# Корутина, возвращающая объект класса UserManager.
async def get_user_manager(user_db=Depends(get_user_db)):
    yield UserManager(user_db)
```

5. Объект класса `FastAPIUsers`, связывающий объект класса `UserManager` и бэкенд аутентификации.

```
# Файл app/core/user.py

fastapi_users = FastAPIUsers[User, int](
    get_user_manager,
    [auth_backend],
)
```

6. Методы класса `FastAPIUsers`, которые будут применены в системе инъекции зависимостей (Dependency Injection) для получения текущего пользователя при выполнении запросов, а также для разграничения доступа:

```
# Файл app/core/user.py

current_user = fastapi_users.current_user(active=True)
current_superuser = fastapi_users.current_user(active=True, superuser=True)
```

7. Роутеры в файле *app/api/endpoints/user.py*:

```python
# Файл app/api/endpoints/user.py

from fastapi import APIRouter

from app.core.user import auth_backend, fastapi_users
from app.schemas.user import UserCreate, UserRead, UserUpdate

router = APIRouter()

router.include_router(
    # В роутер аутентификации
    # должен быть передан объект бэкенда аутентификации.
    fastapi_users.get_auth_router(auth_backend),
    prefix='/auth/jwt',
    tags=['auth'],
)

# Подключаем роутер для регистрации, импортированный из fastapi_users:
router.include_router(
    fastapi_users.get_register_router(UserRead, UserCreate),
    prefix='/auth',
    tags=['auth'],
)
```

```
users_router = fastapi_users.get_users_router(UserRead, UserUpdate)
users_router.routes = [
    # У каждого эндпоинта в библиотеке FastAPI User есть "личное имя",
    # оно устанавливается в параметре name эндпоинта.
    # По этому имени можно обратиться к эндпоинту или идентифицировать его.
    # Имя эндпоинта для удаления пользователей - 'users:delete_user'.
    # Заново переподключаем к роутеру все эндпоинты,
    # исключив эндпоинт для удаления пользователя.
    route for route in users_router.routes if route.name != 'users:delete_user'
]
# Подключаем изменённый роутер по старому адресу.
router.include_router(
    users_router,
    prefix='/users',
    tags=['users'],
)
```

Подключение роутера пользователей к главному роутеру в *app/api/routers.py*:

```
# Файл app/api/routers.py
from fastapi import APIRouter

from app.api.endpoints import (
    ..., user_router
)
```

```
...
main_router.include_router(user_router)
```

Работа с пользователями

Подключение модели пользователя к другой модели проекта через поле `ForeignKey`:

```
from sqlalchemy import ForeignKey, Integer
from sqlalchemy.orm import Mapped, mapped_column

from app.core.db import Base


class TheModel(Base):
    ...
    user_id: Mapped[int] = mapped_column(
        Integer,
        ForeignKey('user.id'),
        name='fk_themodel_user_id_user',
    )
```

Получение объекта текущего пользователя:

```python
from typing import Annotated

from app.core.user import current_user
from app.models import User


@router.get('/the_path', ...)
def the_view(
    ...
    user: Annotated[User, Depends(current_user)],
): ...
```

Ограничение доступа:

```python
from app.core.user import current_superuser


@router.get(
    '/the_path',
    ...
    # Значение dependencies - всегда список.
    dependencies=[Depends(current_superuser)],
```

```
)
def the_view(...): ...
```

Здесь аргумент `dependencies` используется для определения зависимостей, которые должны быть выполнены перед выполнением функции-обработчика `the_view()`.

Автоматическое создание суперпользователя

Изменяем настройки проекта:

```
from pydantic import EmailStr
from pydantic_settings import BaseSettings, SettingsConfigDict


class Settings(BaseSettings):
        ...
    first_superuser_email: EmailStr | None = None
    first_superuser_password: str | None = None
        ...
```

Добавляем код для создания объекта пользователя в *app/core/init_db.py*:

```
import contextlib

from fastapi_users.exceptions import UserAlreadyExists
from pydantic import EmailStr
```

```python
from app.core.config import settings
from app.core.db import get_async_session
from app.core.user import get_user_db, get_user_manager
from app.schemas.user import UserCreate

# Превращаем асинхронные генераторы в асинхронные менеджеры контекста.
get_async_session_context = contextlib.asynccontextmanager(get_async_session)
get_user_db_context = contextlib.asynccontextmanager(get_user_db)
get_user_manager_context = contextlib.asynccontextmanager(get_user_manager)


# Корутина, создающая юзера с переданным email и паролем.
# Возможно создание суперюзера при передаче аргумента is_superuser=True.
async def create_user(
        email: EmailStr, password: str, is_superuser: bool = False
):
    try:
        # Получение объекта асинхронной сессии.
        async with get_async_session_context() as session:
            # Получение объекта класса SQLAlchemyUserDatabase.
            async with get_user_db_context(session) as user_db:
                # Получение объекта класса UserManager.
                async with get_user_manager_context(user_db) as user_manager:
                    # Создание пользователя.
```

```
                    await user_manager.create(
                        UserCreate(
                            email=email,
                            password=password,
                            is_superuser=is_superuser
                        )
                    )
        # В случае, если такой пользователь уже есть, ничего не предпринимать.
        except UserAlreadyExists:
            pass


# Корутина, проверяющая, указаны ли в настройках данные для суперюзера.
# Если да, то вызывается корутина create_user для создания суперпользователя.
async def create_first_superuser():
    if (settings.first_superuser_email is not None
            and settings.first_superuser_password is not None):
        await create_user(
            email=settings.first_superuser_email,
            password=settings.first_superuser_password,
            is_superuser=True,
        )
```

Чтобы при запуске приложения автоматически создавался суперпользователь — в момент старта приложения надо вызвать корутину `create_first_superuser()`. В FastAPI для этого применяется такой подход:

- Создаётся асинхронная функция, в ней описываются все действия, которые будут выполняться перед запуском FastAPI-приложения и сразу после завершения его работы. Эту функцию традиционно называют `lifespan()`. Она принимает на вход объект приложения.

  В нашем примере в функции `lifespan()` будет вызываться корутина `create_first_superuser()`.
- Функция `lifespan()` оборачивается в декоратор `@asynccontextmanager` из встроенной библиотеки contextlib.
- Функция `lifespan()` передаётся в аргумент `lifespan` объекта приложения.

В итоге:

- при старте приложения первым делом выполнятся те инструкции, которые описаны в `lifespan()` до ключевого слова `yield`;
- при остановке приложения будут выполнены инструкции, описанные после ключевого слова `yield` — и лишь после этого приложение завершит работу.

Функция `lifespan()` не должна возвращать никаких значений: инструкция `yield` должна быть пустой.

Код разместим в *app/main.py*:

```
# Потребуется дополнительный импорт:
from contextlib import asynccontextmanager


...
# При старте приложения запускаем корутину create_first_superuser.
@asynccontextmanager
async def lifespan(app: FastAPI):
    # Всё, что указано выше yield, выполняется до запуска приложения.
```

```
    await create_first_superuser()

    # Lifespan-функция обязана вызывать yield,
    # но не должна возвращать никаких значений.
    yield

    # Все инструкции, описанные после yield,
    # выполняется перед завершением работы приложения.
    # В нашем случае ничего выполнять не нужно, но можно и пошалить:
    print('И все эти мгновения исчезнут во времени, как слёзы под дождём.')
```

Я Практикум