Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления» (ИУ)

КАФЕДРА «Информационная безопасность» (ИУ8)

Отчёт

по лабораторной работе № 7-8
по дисциплине «Программирование на Python»

Тема: «FastAPI, разработка API приложение»

Выполнил: Бородин Г.
студент группы ИУ8-13М

Проверил: Зотов М.

г. Москва, 2025 г.

1. ЦЕЛЬ РАБОТЫ

Реализовать API для функционала приложения по варианту

2. ПОСТАНОВКА ЗАДАЧИ

- Ваше API использует асинхронное подключение к PostgreSQL через asyncpg
- Ваши API строит ORM-модели на SQLAlchemy 2.0
- Ваш API применяет валидацию и сериализацию через Pydantic (v2)
- Ваш REST API доступен через FastAPI
- Ваш API запускается через uvicorn (например: uvicorn main:app --reload), либо поднимается в докер контейнере

Описание

Асинхронное API для создания, обновления и отслеживания задач с возможностью назначать исполнителей и устанавливать сроки.

Основные сущности:

user (id, name, email)
task (id, title, description, status, due_date, assignee_id → User)

Эндпоинты:

Создание задачи и пользователей
Обновление задачи и пользователей
фильтрация по статусу
получение задач пользователя.

3. ХОД РАБОТЫ

ИСПОЛЬЗУЕТСЯ В ДАННОЙ РАБОТЕ: Python 3.11; FastAPI для реализации REST API; Uvicorn для запуска ASGI-приложения; PostgreSQL как СУБД; асинхронный драйвер asyncpg; SQLAlchemy 2.0 (AsyncEngine/AsyncSession) для ORM-моделей и запросов; Pydantic v2 для валидации входных данных и сериализации ответов; Docker / docker-compose для развёртывания приложения и базы данных в контейнерах.

МОЙ ХОД РАБОТЫ:

ШАГ 1. Анализ требований и проектирование сущностей.

Определены сущности предметной области: User (id, name, email) и Task (id, title, description, status, due_date, assignee_id). Определены необходимые операции: создание и обновление пользователей и задач, фильтрация задач по статусу, получение задач конкретного пользователя.

ШАГ 2. Реализация ORM-моделей SQLAlchemy 2.0.

Созданы ORM-классы UserORM и TaskORM, определены поля, типы и ограничения (например, уникальность email). Для статуса задачи используется перечисление StatusEnum, поле due_date хранится как timezone-aware datetime.

ШАГ 3. Реализация Pydantic-схем.

Созданы схемы для запросов и ответов: UserCreate/UserGet, TaskCreate/TaskUpdate/TaskGet. Реализована валидация дат: приведение due_date к timezone-aware виду и проверка, что срок выполнения находится в будущем.

ШАГ 4. Настройка асинхронного подключения к Postgres.

Создан AsyncEngine и фабрика сессий AsyncSessionLocal через create_async_engine и async_sessionmaker. Это обеспечивает выполнение SQLAlchemy-запросов в асинхронном режиме.

ШАГ 5. Реализация CRUD-операций и роутеров FastAPI.

Созданы функции операций (create/update/get) для пользователей и задач с использованием AsyncSession. Реализованы роутеры /users и /tasks, которые принимают Pydantic-модели на вход и возвращают сериализованные ответы.

ШАГ 6. Инициализация приложения и запуск.

В main.py реализован lifespan, который создаёт таблицы при старте приложения (Base.metadata.create_all). Подготовлены Dockerfile и docker-compose для запуска приложения вместе с PostgreSQL.

4. ЗАКЛЮЧЕНИЕ

В ходе лабораторной работы были изучены принципы разработки асинхронного REST API на FastAPI и организация работы с базой данных PostgreSQL через asyncpg и SQLAlchemy 2.0 в асинхронном режиме. Были реализованы ORM-модели пользователей и задач, Pydantic-схемы для валидации и сериализации, а также набор эндпоинтов для создания/обновления сущностей, фильтрации задач по статусу и получения задач пользователя. Приложение запускается через uvicorn и может быть развёрнуто в Docker-окружении через docker-compose.

5. ПРИЛОЖЕНИЕ

Реализованный код располагается по следующей ссылке: https://github.com/zezOtik/bmstu--iu8--python/tree/feature/Borodin/Laba7

Листинг1– ORM-модели SQLAlchemy 2.0: определение классов UserORM и TaskORM, базового класса Base, перечисления StatusEnum и функций получения текущего UTC-времени.

```
from sqlalchemy.orm import DeclarativeBase, Mapped, mapped_column
from sqlalchemy import String, Text, DateTime, Integer, ForeignKey, Enum
from sqlalchemy.ext.asyncio import AsyncAttrs
from datetime import datetime
from enum import Enum as PyEnum
from zoneinfo import ZoneInfo


def utc_now():
    return datetime.now(ZoneInfo("UTC"))


class StatusEnum(PyEnum):
    pending = "pending"
    in_progress = "in_progress"
    done = "done"


class Base(AsyncAttrs, DeclarativeBase):
    pass


class UserORM(Base):
```

```
    __tablename__ = "user"

    id: Mapped[int] = mapped_column(Integer, primary_key=True,
autoincrement=True)
    name: Mapped[str] = mapped_column(String(100))
    email: Mapped[str] = mapped_column(String(100), unique=True)



class TaskORM(Base):
    __tablename__ = "task"

    id: Mapped[int] = mapped_column(Integer, primary_key=True,
autoincrement=True)
    title: Mapped[str] = mapped_column(String(200))
    description: Mapped[str] = mapped_column(Text, nullable=True)
    status: Mapped[StatusEnum] = mapped_column(Enum(StatusEnum),
default=StatusEnum.pending)
    due_date: Mapped[datetime] = mapped_column(DateTime(timezone=True))
    assignee_id: Mapped[int] = mapped_column(ForeignKey("user.id"))
```

Листинг2– Pydantic-схемы: модели запросов/ответов для пользователей и задач, валидация email, проверка due_date (timezone-aware и срок в будущем), схемы для обновления задачи и списков.

```
from pydantic import BaseModel, Field, EmailStr, field_validator
from typing import Optional, List
from datetime import datetime
from zoneinfo import ZoneInfo
from enum import Enum
```

```
def ensure_utc(dt: datetime) -> datetime:
    if dt.tzinfo is None:
        return dt.replace(tzinfo=ZoneInfo("UTC"))
    return dt


class StatusEnum(str, Enum):
    pending = "pending"
    in_progress = "in_progress"
    done = "done"


class UserBase(BaseModel):
    name: str = Field(..., min_length=1, max_length=100)
    email: EmailStr


class UserCreate(UserBase):
    pass


class UserGet(UserBase):
    id: int

    model_config = {"from_attributes": True}


class TaskBase(BaseModel):
    title: str = Field(..., min_length=1, max_length=200)
```

```
    description: Optional[str] = None
    status: StatusEnum = StatusEnum.pending
    due_date: datetime
    assignee_id: int

    @field_validator("due_date", mode="before")
    @classmethod
    def validate_due_date(cls, v):
        if isinstance(v, str):
            v = datetime.fromisoformat(v)
        return ensure_utc(v)

    @field_validator("due_date")
    @classmethod
    def due_date_must_be_future(cls, v):
        now = datetime.now(ZoneInfo("UTC"))
        if v < now:
            raise ValueError("due_date must be in the future")
        return v


class TaskCreate(TaskBase):
    pass


class TaskUpdate(BaseModel):
    title: Optional[str] = None
    description: Optional[str] = None
    status: Optional[StatusEnum] = None
```

```
    due_date: Optional[datetime] = None
    assignee_id: Optional[int] = None

    @field_validator("due_date", mode="before")
    @classmethod
    def validate_due_date(cls, v):
        if v is None:
            return v
        if isinstance(v, str):
            v = datetime.fromisoformat(v)
        return ensure_utc(v)


class TaskGet(TaskBase):
    id: int

    model_config = {"from_attributes": True}


class TaskListGet(BaseModel):
    tasks: List[TaskGet]


class TasksByUser(BaseModel):
    user_id: int
```

Листинг 3– Подключение к БД: создание асинхронного SQLAlchemy engine по DATABASE_URL и фабрики сессий AsyncSessionLocal.

```
import os
from sqlalchemy.ext.asyncio import create_async_engine, async_sessionmaker
```

```
from sqlalchemy.pool import NullPool


DATABASE_URL = os.getenv(
    "DATABASE_URL",
    "postgresql+asyncpg://postgres:postgres@localhost:5432/postgres"
)


engine = create_async_engine(DATABASE_URL, echo=True,
poolclass=NullPool)
AsyncSessionLocal = async_sessionmaker(engine, expire_on_commit=False)
```

Листинг 4– Операции с задачами: создание задачи, обновление по id, выборка задач по статусу и по пользователю через async SQLAlchemy.

```
from sqlalchemy import select, update
from app.operations.base import AsyncSessionLocal
from app.models.database_models import TaskORM, StatusEnum
from app.models.store_models import TaskCreate, TaskUpdate



async def create_task(task_in: TaskCreate):
    async with AsyncSessionLocal() as session:
        task = TaskORM(**task_in.model_dump())
        session.add(task)
        await session.commit()
        await session.refresh(task)
        return task



async def update_task(task_id: int, task_in: TaskUpdate):
    async with AsyncSessionLocal() as session:
```

```
        values = {k: v for k, v in task_in.model_dump().items() if v is not None}
        stmt = (
            update(TaskORM)
            .where(TaskORM.id == task_id)
            .values(**values)
            .returning(TaskORM)
        )
        result = await session.execute(stmt)
        await session.commit()
        return result.scalar_one()



async def get_tasks_by_status(status: str):
    async with AsyncSessionLocal() as session:
        stmt = select(TaskORM).where(TaskORM.status == StatusEnum(status))
        result = await session.execute(stmt)
        return result.scalars().all()



async def get_tasks_by_user(user_id: int):
    async with AsyncSessionLocal() as session:
        stmt = select(TaskORM).where(TaskORM.assignee_id == user_id)
        result = await session.execute(stmt)
        return result.scalars().all()
```

Листинг 5– Операции с пользователями: создание пользователя с обработкой конфликта уникального email, обновление пользователя и получение пользователя по id.

```
from sqlalchemy import select, update
from sqlalchemy.exc import IntegrityError
```

```
from app.operations.base import AsyncSessionLocal
from app.models.database_models import UserORM
from app.models.store_models import UserCreate


async def create_user(user_in: UserCreate):
    async with AsyncSessionLocal() as session:
        user = UserORM(**user_in.model_dump())
        session.add(user)
        try:
            await session.commit()
            await session.refresh(user)
        except IntegrityError:
            await session.rollback()
            raise ValueError("User with this email already exists")
        return user


async def update_user(user_id: int, user_in: UserCreate):
    async with AsyncSessionLocal() as session:
        stmt = (
            update(UserORM)
            .where(UserORM.id == user_id)
            .values(**user_in.model_dump())
            .returning(UserORM)
        )
        result = await session.execute(stmt)
        await session.commit()
        return result.scalar_one()
```

```
async def get_user(user_id: int):
    async with AsyncSessionLocal() as session:
        stmt = select(UserORM).where(UserORM.id == user_id)
        result = await session.execute(stmt)
        return result.scalar_one()
```

Листинг 6– Роутер задач FastAPI (/tasks): эндпоинты создания/обновления задачи, фильтрации по статусу и получения задач пользователя.

```
from fastapi import APIRouter, Query
from typing import List, Optional
from app.models.store_models import TaskCreate, TaskUpdate, TaskGet,
TasksByUser
from app.operations.task_ops import (
    create_task,
    update_task,
    get_tasks_by_status,
    get_tasks_by_user,
)

router = APIRouter(prefix="/tasks", tags=["tasks"])

@router.post("/", response_model=TaskGet)
async def create_new_task(task: TaskCreate):
    return await create_task(task)

@router.put("/{task_id}", response_model=TaskGet)
async def update_existing_task(task_id: int, task: TaskUpdate):
```

```
    return await update_task(task_id, task)


@router.get("/by-status", response_model=List[TaskGet])
async def read_tasks_by_status(status: str = Query(...)):
    return await get_tasks_by_status(status)


@router.get("/user/{user_id}", response_model=List[TaskGet])
async def read_tasks_for_user(user_id: int):
    return await get_tasks_by_user(user_id)
```

Листинг 7– Роутер пользователей FastAPI (/users): эндпоинты создания, обновления и получения пользователя по id.

```
from fastapi import APIRouter, Depends
from app.models.store_models import UserCreate, UserGet
from app.operations.user_ops import create_user, update_user, get_user


router = APIRouter(prefix="/users", tags=["users"])


@router.post("/", response_model=UserGet)
async def create_new_user(user: UserCreate):
    return await create_user(user)


@router.put("/{user_id}", response_model=UserGet)
async def update_existing_user(user_id: int, user: UserCreate):
    return await update_user(user_id, user)


@router.get("/{user_id}", response_model=UserGet)
async def read_user(user_id: int):
    return await get_user(user_id)
```

Листинг 8– Инициализация приложения: FastAPI(lifespan=...), создание таблиц при старте, подключение роутеров пользователей и задач.

```
from fastapi import FastAPI
from contextlib import asynccontextmanager
from app.operations.base import engine
from app.models.database_models import Base

@asynccontextmanager
async def lifespan(app: FastAPI):
    async with engine.begin() as conn:
        await conn.run_sync(Base.metadata.create_all)
    yield
    await engine.dispose()

app = FastAPI(lifespan=lifespan)

from app.router.user import router as user_router
from app.router.task import router as task_router

app.include_router(user_router)
app.include_router(task_router)
```

Листинг 9– Dockerfile: сборка образа приложения, установка зависимостей и запуск uvicorn внутри контейнера.

```
FROM python:3.11-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Листинг 10– *docker-compose:* запуск PostgreSQL и FastAPI-приложения, настройка переменных окружения и ожидание готовности БД через healthcheck.

```
version: '3.9'


services:
  postgres:
    image: postgres:13
    container_name: postgres_container
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres
      POSTGRES_DB: postgres
      PGDATA: /var/lib/postgresql/data/pgdata
    ports:
      - "5432:5432"
    volumes:
      - ./pgdata:/var/lib/postgresql/data/pgdata
    command: >
      postgres -c max_connections=1000
               -c shared_buffers=256MB
               -c effective_cache_size=768MB
               -c maintenance_work_mem=64MB
               -c checkpoint_completion_target=0.7
               -c wal_buffers=16MB
               -c default_statistics_target=100
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U postgres -d postgres"]
      interval: 3s
      timeout: 5s
      retries: 10
      start_period: 10s
```

```
    restart: unless-stopped


  app:
    build: .
    container_name: fastapi_app
    ports:
      - "8000:8000"
    environment:
      DATABASE_URL:
postgresql+asyncpg://postgres:postgres@postgres:5432/postgres
    depends_on:
      postgres:
        condition: service_healthy
    command: uvicorn app.main:app --host 0.0.0.0 --port 8000
    volumes:
      - .:/app
    restart: unless-stopped
```

Листинг 11– *requirements.txt:* список Python-зависимостей проекта (FastAPI, Pydantic, SQLAlchemy, asyncpg, uvicorn).

```
fastapi==0.110.2
pydantic[email]==2.7.0
SQLAlchemy==2.0.30
uvicorn==0.29.0
asyncpg==0.29.0
```