МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ
«БРЕСТСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»
ФАКУЛЬТЕТ ЭЛЕКТРОННО-ИНФОРМАЦИОННЫХ СИСТЕМ

Кафедра интеллектуальных информационных технологий

Отчёт по лабораторной работе №5

Специальность ПО11

Выполнил
С. С. Жватель
студент группы ПО11

Проверил
А. А. Крощенко
ст. преп. кафедры ИИТ,
12.04.2025 г.

Брест 2025

Цель работы: приобрести практические навыки разработки API и баз данных

Общее задание

1. Реализовать базу данных из не менее 5 таблиц на заданную тематику. При реализации продумать типизацию полей и внешние ключи в таблицах;
2. Визуализировать разработанную БД с помощью схемы, на которой отображены все таблицы и связи между ними (пример, схема на рис. 1);
3. На языке Python с использованием SQLAlchemy реализовать подключение к БД;
4. Реализовать основные операции с данными (выборку, добавление, удаление, модификацию);
5. Для каждой реализованной операции с использованием FastAPI реализовать отдельный эндпойнт;

Базу данные можно реализовать в любой СУБД (MySQL, PostgreSQL, SQLite и др.)

Задание. База данных Справочное бюро ж/д вокзала

Выполнение:

Код программы:

```
"""Railway Management API.

This module provides a FastAPI-based API for managing railway-related entities
such as trains, routes, stations, employees, and schedules. It uses SQLAlchemy
for database operations with a SQLite backend and Pydantic for request/response
validation.
"""

from datetime import datetime
from typing import List

from fastapi import Depends, FastAPI, HTTPException
from pydantic import BaseModel
from sqlalchemy import (
    Column,
    DateTime,
    Float,
    ForeignKey,
    Integer,
    String,
    create_engine,
)
from sqlalchemy.orm import declarative_base, relationship, sessionmaker

# Database configuration (SQLite)
DATABASE_URL = "sqlite:///./railway.db"
engine = create_engine(DATABASE_URL)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
Base = declarative_base()


# SQLAlchemy Models
# pylint: disable=too-few-public-methods
class Train(Base):
    """SQLAlchemy model for a train entity.
```

```
    Represents a train with attributes like train number, model, and capacity.
    Associated with schedules via a relationship.
    """

    __tablename__ = "trains"
    id = Column(Integer, primary_key=True, index=True)
    train_number = Column(String(20), nullable=False, unique=True)
    model = Column(String(100), nullable=False)
    capacity = Column(Integer, nullable=False)
    schedules = relationship("Schedule", back_populates="train")


# pylint: disable=too-few-public-methods
class Route(Base):
    """SQLAlchemy model for a route entity.

    Represents a route with start and end stations, distance, and associated schedules
    and stations.
    """

    __tablename__ = "routes"
    id = Column(Integer, primary_key=True, index=True)
    route_number = Column(String(10), nullable=False, unique=True)
    start_station = Column(String(100), nullable=False)
    end_station = Column(String(100), nullable=False)
    distance_km = Column(Float, nullable=False)
    schedules = relationship("Schedule", back_populates="route")
    stations = relationship("Station", secondary="route_stations", back_populates="routes")


# pylint: disable=too-few-public-methods
class Station(Base):
    """SQLAlchemy model for a station entity.

    Represents a railway station with a name and geographic coordinates.
    Associated with routes via a relationship.
    """

    __tablename__ = "stations"
    id = Column(Integer, primary_key=True, index=True)
    name = Column(String(100), nullable=False, unique=True)
    latitude = Column(Float, nullable=False)
    longitude = Column(Float, nullable=False)
    routes = relationship("Route", secondary="route_stations", back_populates="stations")


# pylint: disable=too-few-public-methods
class Employee(Base):
```

```
    """SQLAlchemy model for an employee entity.

    Represents an employee with personal details, license number, and position.
    Associated with schedules via a relationship.
    """

    __tablename__ = "employees"
    id = Column(Integer, primary_key=True, index=True)
    first_name = Column(String(50), nullable=False)
    last_name = Column(String(50), nullable=False)
    license_number = Column(String(20), nullable=False, unique=True)
    position = Column(String(50), nullable=False)
    schedules = relationship("Schedule", back_populates="employee")


# pylint: disable=too-few-public-methods
class Schedule(Base):
    """SQLAlchemy model for a schedule entity.

    Represents a train schedule with route, train, employee, and departure time.
    """

    __tablename__ = "schedules"
    id = Column(Integer, primary_key=True, index=True)
    route_id = Column(Integer, ForeignKey("routes.id"), nullable=False)
    train_id = Column(Integer, ForeignKey("trains.id"), nullable=False)
    employee_id = Column(Integer, ForeignKey("employees.id"), nullable=False)
    departure_time = Column(DateTime, nullable=False)
    route = relationship("Route", back_populates="schedules")
    train = relationship("Train", back_populates="schedules")
    employee = relationship("Employee", back_populates="schedules")


# pylint: disable=too-few-public-methods
class RouteStation(Base):
    """SQLAlchemy model for the route-station association.

    Represents the many-to-many relationship between routes and stations with stop order.
    """

    __tablename__ = "route_stations"
    route_id = Column(Integer, ForeignKey("routes.id"), primary_key=True)
    station_id = Column(Integer, ForeignKey("stations.id"), primary_key=True)
    stop_order = Column(Integer, nullable=False)


# Create database tables
Base.metadata.create_all(bind=engine)
```

```
# Initialize FastAPI
app = FastAPI()


# Pydantic Models for Validation
# pylint: disable=too-few-public-methods
class TrainBase(BaseModel):
    """Base Pydantic model for train data validation.

    Attributes:
        train_number: Unique identifier for the train.
        model: Train model name.
        capacity: Passenger capacity of the train.
    """

    train_number: str
    model: str
    capacity: int


# pylint: disable=too-few-public-methods
class TrainCreate(TrainBase):
    """Pydantic model for creating a train.

    Inherits all attributes from TrainBase for train creation.
    """


# pylint: disable=too-few-public-methods
class TrainResponse(TrainBase):
    """Pydantic model for train response data, including the train's ID."""

    id: int

    class Config:
        """Sets true"""

        from_attributes = True


# pylint: disable=too-few-public-methods
class RouteBase(BaseModel):
    """Base Pydantic model for route data validation.

    Attributes:
        route_number: Unique identifier for the route.
        start_station: Starting station name.
        end_station: Ending station name.
        distance_km: Distance of the route in kilometers.
```

```
    """

    route_number: str
    start_station: str
    end_station: str
    distance_km: float


# pylint: disable=too-few-public-methods
class RouteCreate(RouteBase):
    """Pydantic model for creating a route.

    Inherits all attributes from RouteBase for route creation.
    """


# pylint: disable=too-few-public-methods
class RouteResponse(RouteBase):
    """Pydantic model for route response data, including the route's ID."""

    id: int

    class Config:
        """Sets true"""

        from_attributes = True


# pylint: disable=too-few-public-methods
class StationBase(BaseModel):
    """Base Pydantic model for station data validation.

    Attributes:
        name: Name of the station.
        latitude: Geographic latitude of the station.
        longitude: Geographic longitude of the station.
    """

    name: str
    latitude: float
    longitude: float


# pylint: disable=too-few-public-methods
class StationCreate(StationBase):
    """Pydantic model for creating a station.

    Inherits all attributes from StationBase for station creation.
    """
```

```python
# pylint: disable=too-few-public-methods
class StationResponse(StationBase):
    """Pydantic model for station response data, including the station's ID."""

    id: int

    class Config:
        """Sets true"""

        from_attributes = True


# pylint: disable=too-few-public-methods
class EmployeeBase(BaseModel):
    """Base Pydantic model for employee data validation.

    Attributes:
        first_name: Employee's first name.
        last_name: Employee's last name.
        license_number: Unique license number of the employee.
        position: Employee's job position.
    """

    first_name: str
    last_name: str
    license_number: str
    position: str


# pylint: disable=too-few-public-methods
class EmployeeCreate(EmployeeBase):
    """Pydantic model for creating an employee.

    Inherits all attributes from EmployeeBase for employee creation.
    """


# pylint: disable=too-few-public-methods
class EmployeeResponse(EmployeeBase):
    """Pydantic model for employee response data, including the employee's ID."""

    id: int

    class Config:
        """Sets true"""

        from_attributes = True
```

```
# pylint: disable=too-few-public-methods
class ScheduleBase(BaseModel):
    """Base Pydantic model for schedule data validation.

    Attributes:
        route_id: Identifier of the associated route.
        train_id: Identifier of the associated train.
        employee_id: Identifier of the associated employee.
        departure_time: Scheduled departure time.
    """

    route_id: int
    train_id: int
    employee_id: int
    departure_time: datetime


# pylint: disable=too-few-public-methods
class ScheduleCreate(ScheduleBase):
    """Pydantic model for creating a schedule.

    Inherits all attributes from ScheduleBase for schedule creation.
    """


# pylint: disable=too-few-public-methods
class ScheduleResponse(ScheduleBase):
    """Pydantic model for schedule response data, including the schedule's ID."""

    id: int

    class Config:
        """Sets true"""

        from_attributes = True


def get_db():
    """Provide a database session for dependency injection.

    Yields:
        SessionLocal: An SQLAlchemy session for database operations.
    """
    db = SessionLocal()
    try:
        yield db
    finally:
```

```
        db.close()


# Train Endpoints
@app.post("/trains/", response_model=TrainResponse)
def create_train(train: TrainCreate, db: SessionLocal = Depends(get_db)):
    """Create a new train in the database.

    Args:
        train: Pydantic model containing train data.
        db: Database session.

    Returns:
        TrainResponse: The created train's details.
    """
    db_train = Train(**train.dict())
    db.add(db_train)
    db.commit()
    db.refresh(db_train)
    return db_train


@app.get("/trains/", response_model=List[TrainResponse])
def get_trains(db: SessionLocal = Depends(get_db)):
    """Retrieve all trains from the database.

    Args:
        db: Database session.

    Returns:
        List[TrainResponse]: List of all trains.
    """
    return db.query(Train).all()


@app.get("/trains/{train_id}", response_model=TrainResponse)
def get_train(train_id: int, db: SessionLocal = Depends(get_db)):
    """Retrieve a specific train by ID.

    Args:
        train_id: ID of the train to retrieve.
        db: Database session.

    Returns:
        TrainResponse: Details of the requested train.

    Raises:
        HTTPException: If the train is not found.
    """
```

```
    train = db.query(Train).filter(Train.id == train_id).first()
    if train is None:
        raise HTTPException(status_code=404, detail="Train not found")
    return train


@app.put("/trains/{train_id}", response_model=TrainResponse)
def update_train(train_id: int, train: TrainCreate, db: SessionLocal = Depends(get_db)):
    """Update an existing train's details.

    Args:
        train_id: ID of the train to update.
        train: Pydantic model containing updated train data.
        db: Database session.

    Returns:
        TrainResponse: Updated train details.

    Raises:
        HTTPException: If the train is not found.
    """
    db_train = db.query(Train).filter(Train.id == train_id).first()
    if db_train is None:
        raise HTTPException(status_code=404, detail="Train not found")
    for key, value in train.dict().items():
        setattr(db_train, key, value)
    db.commit()
    db.refresh(db_train)
    return db_train


@app.delete("/trains/{train_id}")
def delete_train(train_id: int, db: SessionLocal = Depends(get_db)):
    """Delete a train from the database.

    Args:
        train_id: ID of the train to delete.
        db: Database session.

    Returns:
        dict: Confirmation message.

    Raises:
        HTTPException: If the train is not found.
    """
    train = db.query(Train).filter(Train.id == train_id).first()
    if train is None:
        raise HTTPException(status_code=404, detail="Train not found")
    db.delete(train)
```

```
    db.commit()
    return {"message": "Train deleted"}


# Route Endpoints
@app.post("/routes/", response_model=RouteResponse)
def create_route(route: RouteCreate, db: SessionLocal = Depends(get_db)):
    """Create a new route in the database.

    Args:
        route: Pydantic model containing route data.
        db: Database session.

    Returns:
        RouteResponse: The created route's details.
    """
    db_route = Route(**route.dict())
    db.add(db_route)
    db.commit()
    db.refresh(db_route)
    return db_route


@app.get("/routes/", response_model=List[RouteResponse])
def get_routes(db: SessionLocal = Depends(get_db)):
    """Retrieve all routes from the database.

    Args:
        db: Database session.

    Returns:
        List[RouteResponse]: List of all routes.
    """
    return db.query(Route).all()


@app.get("/routes/{route_id}", response_model=RouteResponse)
def get_route(route_id: int, db: SessionLocal = Depends(get_db)):
    """Retrieve a specific route by ID.

    Args:
        route_id: ID of the route to retrieve.
        db: Database session.

    Returns:
        RouteResponse: Details of the requested route.

    Raises:
        HTTPException  HTTPException: If the route is not found.
```

```
    """
    route = db.query(Route).filter(Route.id == route_id).first()
    if route is None:
        raise HTTPException(status_code=404, detail="Route not found")
    return route


@app.put("/routes/{route_id}", response_model=RouteResponse)
def update_route(route_id: int, route: RouteCreate, db: SessionLocal = Depends(get_db)):
    """Update an existing route's details.

    Args:
        route_id: ID of the route to update.
        route: Pydantic model containing updated route data.
        db: Database session.

    Returns:
        RouteResponse: Updated route details.

    Raises:
        HTTPException: If the route is not found.
    """
    db_route = db.query(Route).filter(Route.id == route_id).first()
    if db_route is None:
        raise HTTPException(status_code=404, detail="Route not found")
    for key, value in route.dict().items():
        setattr(db_route, key, value)
    db.commit()
    db.refresh(db_route)
    return db_route


@app.delete("/routes/{route_id}")
def delete_route(route_id: int, db: SessionLocal = Depends(get_db)):
    """Delete a route from the database.

    Args:
        route_id: ID of the route to delete.
        db: Database session.

    Returns:
        dict: Confirmation message.

    Raises:
        HTTPException: If the route is not found.
    """
    route = db.query(Route).filter(Route.id == route_id).first()
    if route is None:
        raise HTTPException(status_code=404, detail="Route not found")
```

```
    db.delete(route)
    db.commit()
    return {"message": "Route deleted"}


# Station Endpoints
@app.post("/stations/", response_model=StationResponse)
def create_station(station: StationCreate, db: SessionLocal = Depends(get_db)):
    """Create a new station in the database.

    Args:
        station: Pydantic model containing station data.
        db: Database session.

    Returns:
        StationResponse: The created station's details.
    """
    db_station = Station(**station.dict())
    db.add(db_station)
    db.commit()
    db.refresh(db_station)
    return db_station


@app.get("/stations/", response_model=List[StationResponse])
def get_stations(db: SessionLocal = Depends(get_db)):
    """Retrieve all stations from the database.

    Args:
        db: Database session.

    Returns:
        List[StationResponse]: List of all stations.
    """
    return db.query(Station).all()


@app.get("/stations/{station_id}", response_model=StationResponse)
def get_station(station_id: int, db: SessionLocal = Depends(get_db)):
    """Retrieve a specific station by ID.

    Args:
        station_id: ID of the station to retrieve.
        db: Database session.

    Returns:
        StationResponse: Details of the requested station.

    Raises:
```

```
    HTTPException: If the station is not found.
    """
    station = db.query(Station).filter(Station.id == station_id).first()
    if station is None:
        raise HTTPException(status_code=404, detail="Station not found")
    return station


@app.put("/stations/{station_id}", response_model=StationResponse)
def update_station(station_id: int, station: StationCreate, db: SessionLocal = Depends(get_db)):
    """Update an existing station's details.

    Args:
        station_id: ID of the station to update.
        station: Pydantic model containing updated station data.
        db: Database session.

    Returns:
        StationResponse: Updated station details.

    Raises:
        HTTPException: If the station is not found.
    """
    db_station = db.query(Station).filter(Station.id == station_id).first()
    if db_station is None:
        raise HTTPException(status_code=404, detail="Station not found")
    for key, value in station.dict().items():
        setattr(db_station, key, value)
    db.commit()
    db.refresh(db_station)
    return db_station


@app.delete("/stations/{station_id}")
def delete_station(station_id: int, db: SessionLocal = Depends(get_db)):
    """Delete a station from the database.

    Args:
        station_id: ID of the station to delete.
        db: Database session.

    Returns:
        dict: Confirmation message.

    Raises:
        HTTPException: If the station is not found.
    """
    station = db.query(Station).filter(Station.id == station_id).first()
    if station is None:
```

```
        raise HTTPException(status_code=404, detail="Station not found")
    db.delete(station)
    db.commit()
    return {"message": "Station deleted"}


# Employee Endpoints
@app.post("/employees/", response_model=EmployeeResponse)
def create_employee(employee: EmployeeCreate, db: SessionLocal = Depends(get_db)):
    """Create a new employee in the database.

    Args:
        employee: Pydantic model containing employee data.
        db: Database session.

    Returns:
        EmployeeResponse: The created employee's details.
    """
    db_employee = Employee(**employee.dict())
    db.add(db_employee)
    db.commit()
    db.refresh(db_employee)
    return db_employee


@app.get("/employees/", response_model=List[EmployeeResponse])
def get_employees(db: SessionLocal = Depends(get_db)):
    """Retrieve all employees from the database.

    Args:
        db: Database session.

    Returns:
        List[EmployeeResponse]: List of all employees.
    """
    return db.query(Employee).all()


@app.get("/employees/{employee_id}", response_model=EmployeeResponse)
def get_employee(employee_id: int, db: SessionLocal = Depends(get_db)):
    """Retrieve a specific employee by ID.

    Args:
        employee_id: ID of the employee to retrieve.
        db: Database session.

    Returns:
        EmployeeResponse: Details of the requested employee.
```

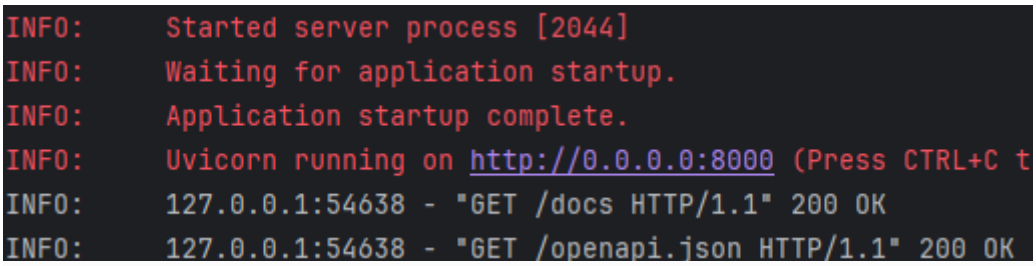
```
    Raises:
        HTTPException: If the employee is not found.
    """
    employee = db.query(Employee).filter(Employee.id == employee_id).first()
    if employee is None:
        raise HTTPException(status_code=404, detail="Employee not found")
    return employee


@app.put("/employees/{employee_id}", response_model=EmployeeResponse)
def update_employee(employee_id: int, employee: EmployeeCreate, db: SessionLocal = Depends(get_db)):
    """Update an existing employee's details.

    Args:
        employee_id: ID of the employee to update.
        employee: Pydantic model containing updated employee data.
        db: Database session.

    Returns:
        EmployeeResponse: Updated employee details.

    Raises:
        HTTPException: If the employee is not found.
    """
    db_employee = db.query(Employee).filter(Employee.id == employee_id).first()
    if db_employee is None:
        raise HTTPException(status_code=404, detail="Employee not found")
    for key, value in employee.dict().items():
        setattr(db_employee, key, value)
    db.commit()
    db.refresh(db_employee)
    return db_employee


@app.delete("/employees/{employee_id}")
def delete_employee(employee_id: int, db: SessionLocal = Depends(get_db)):
    """Delete an employee from the database.

    Args:
        employee_id: ID of the employee to delete.
        db: Database session.

    Returns:
        dict: Confirmation message.

    Raises:
        HTTPException: If the employee is not found.
    """
    employee = db.query(Employee).filter(Employee.id == employee_id).first()
```

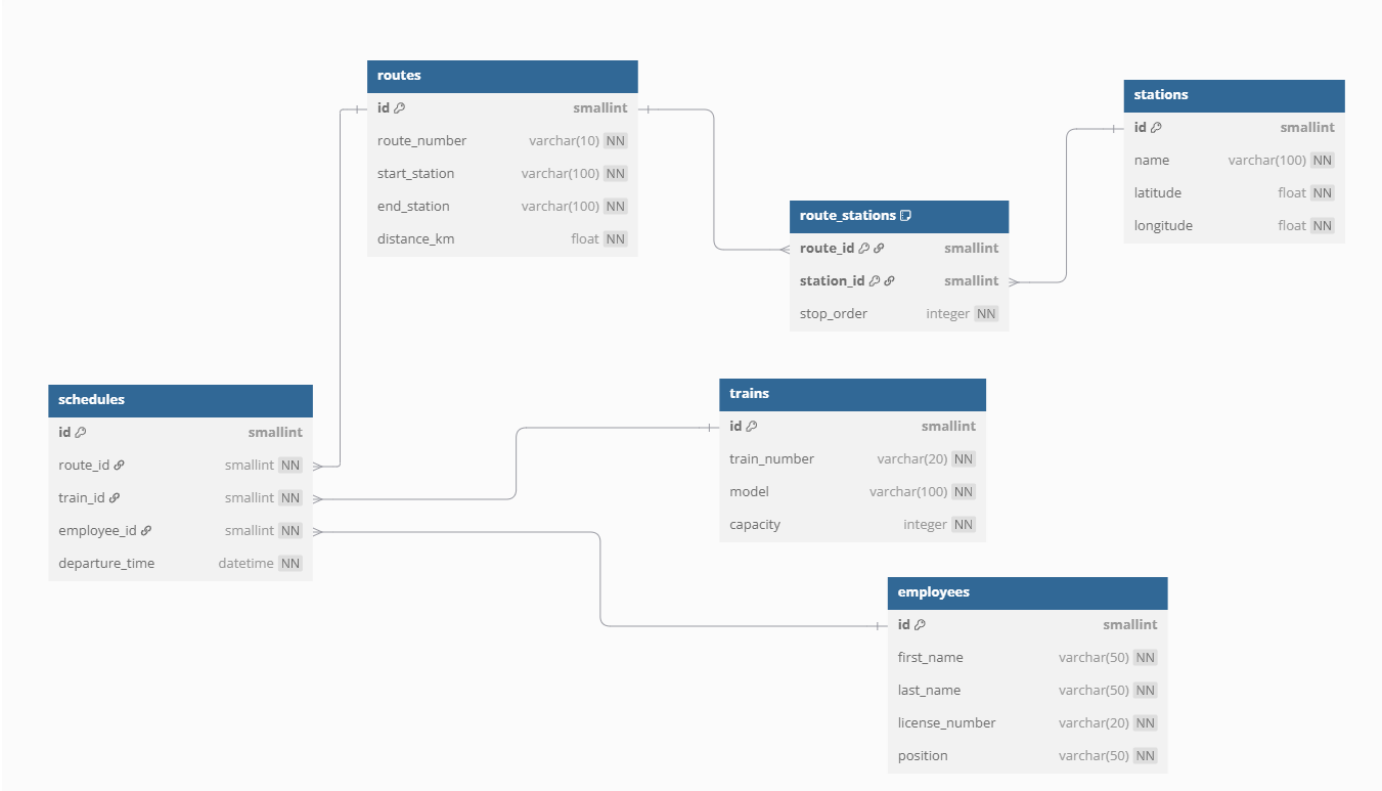
```
    if employee is None:
        raise HTTPException(status_code=404, detail="Employee not found")
    db.delete(employee)
    db.commit()
    return {"message": "Employee deleted"}


# Schedule Endpoints
@app.post("/schedules/", response_model=ScheduleResponse)
def create_schedule(schedule: ScheduleCreate, db: SessionLocal = Depends(get_db)):
    """Create a new schedule in the database.

    Args:
        schedule: Pydantic model containing schedule data.
        db: Database session.

    Returns:
        ScheduleResponse: The created schedule's details.
    """
    db_schedule = Schedule(**schedule.dict())
    db.add(db_schedule)
    db.commit()
    db.refresh(db_schedule)
    return db_schedule


@app.get("/schedules/", response_model=List[ScheduleResponse])
def get_schedules(db: SessionLocal = Depends(get_db)):
    """Retrieve all schedules from the database.

    Args:
        db: Database session.

    Returns:
        List[ScheduleResponse]: List of all schedules.
    """
    return db.query(Schedule).all()


@app.get("/schedules/{schedule_id}", response_model=ScheduleResponse)
def get_schedule(schedule_id: int, db: SessionLocal = Depends(get_db)):
    """Retrieve a specific schedule by ID.

    Args:
        schedule_id: ID of the schedule to retrieve.
        db: Database session.

    Returns:
        ScheduleResponse: Details of the requested schedule.
```

```
    Raises:
        HTTPException: If the schedule is not found.
    """
    schedule = db.query(Schedule).filter(Schedule.id == schedule_id).first()
    if schedule is None:
        raise HTTPException(status_code=404, detail="Schedule not found")
    return schedule


@app.put("/schedules/{schedule_id}", response_model=ScheduleResponse)
def update_schedule(schedule_id: int, schedule: ScheduleCreate, db: SessionLocal = Depends(get_db)):
    """Update an existing schedule's details.

    Args:
        schedule_id: ID of the schedule to update.
        schedule: Pydantic model containing updated schedule data.
        db: Database session.

    Returns:
        ScheduleResponse: Updated schedule details.

    Raises:
        HTTPException: If the schedule is not found.
    """
    db_schedule = db.query(Schedule).filter(Schedule.id == schedule_id).first()
    if db_schedule is None:
        raise HTTPException(status_code=404, detail="Schedule not found")
    for key, value in schedule.dict().items():
        setattr(db_schedule, key, value)
    db.commit()
    db.refresh(db_schedule)
    return db_schedule


@app.delete("/schedules/{schedule_id}")
def delete_schedule(schedule_id: int, db: SessionLocal = Depends(get_db)):
    """Delete a schedule from the database.

    Args:
        schedule_id: ID of the schedule to delete.
        db: Database session.

    Returns:
        dict: Confirmation message.

    Raises:
        HTTPException: If the schedule is not found.
    """
```

```
    schedule = db.query(Schedule).filter(Schedule.id == schedule_id).first()
    if schedule is None:
        raise HTTPException(status_code=404, detail="Schedule not found")
    db.delete(schedule)
    db.commit()
    return {"message": "Schedule deleted"}


if __name__ == "__main__":
    import uvicorn

    uvicorn.run(app, host="0.0.0.0", port=8000)
```

Рисунки с результатами работы программы:





Вывод: закрепил базовые знания Python, API и работе с базой данных при решении практических задач