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
Н. А. Антонюк
студент группы ПО11

Проверил
А. А. Крощенко
ст. преп. кафедры ИИТ,
17.04.2025 г.

Брест 2025

Цель работы: приобрести практические навыки разработки API и баз данных.

Задание:

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.)

Вариант:

База данных-Формула-1

Используемая СУБД-SQLite

Код программы:

```
from fastapi import FastAPI, HTTPException, Depends
from fastapi.middleware.cors import CORSMiddleware
from sqlalchemy import create_engine, Column, Integer, String, ForeignKey, Float, Date
from sqlalchemy.orm import sessionmaker, relationship, Session
from sqlalchemy.ext.declarative import declarative_base
from typing import List, Optional
from pydantic import BaseModel
from datetime import date

# Создание FastAPI приложения
app = FastAPI(title="Formula 1 API")

# Добавление CORS
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],  # Разрешает запросы с любого источника
    allow_credentials=True,
    allow_methods=["*"],  # Разрешает все методы
    allow_headers=["*"],  # Разрешает все заголовки
)

# Создание базы данных SQLite
SQLALCHEMY_DATABASE_URL = "sqlite:///formula1.db"
engine = create_engine(SQLALCHEMY_DATABASE_URL)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
Base = declarative_base()

# Модели базы данных
```

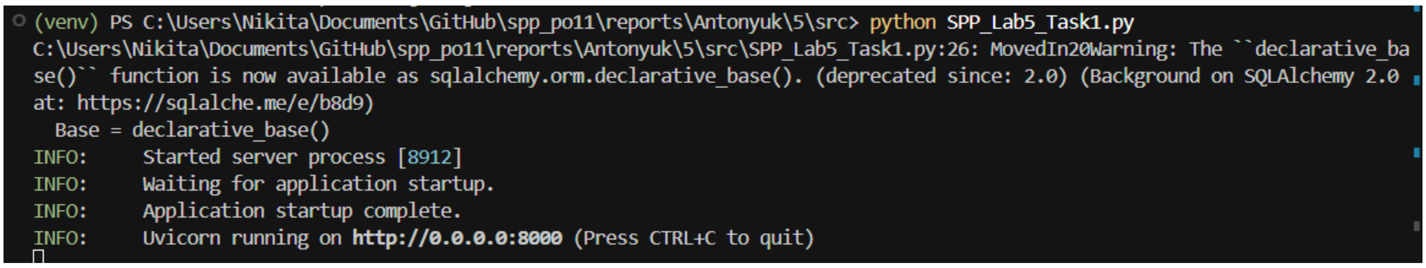
```
class Team(Base):
   __tablename__ = "teams"

   id = Column(Integer, primary_key=True, index=True)
   name = Column(String, unique=True, index=True)
   country = Column(String)
   founded_year = Column(Integer)
   drivers = relationship("Driver", back_populates="team")
   cars = relationship("Car", back_populates="team")

class Driver(Base):
   __tablename__ = "drivers"

   id = Column(Integer, primary_key=True, index=True)
   name = Column(String, index=True)
   nationality = Column(String)
   birth_date = Column(Date)
   team_id = Column(Integer, ForeignKey("teams.id"))
   team = relationship("Team", back_populates="drivers")
   race_results = relationship("RaceResult", back_populates="driver")

class Car(Base):
   __tablename__ = "cars"

   id = Column(Integer, primary_key=True, index=True)
   model = Column(String)
   year = Column(Integer)
   team_id = Column(Integer, ForeignKey("teams.id"))
   team = relationship("Team", back_populates="cars")

class Circuit(Base):
   __tablename__ = "circuits"

   id = Column(Integer, primary_key=True, index=True)
   name = Column(String, unique=True, index=True)
   country = Column(String)
   length = Column(Float)
   races = relationship("Race", back_populates="circuit")

class Race(Base):
   __tablename__ = "races"

   id = Column(Integer, primary_key=True, index=True)
   name = Column(String)
   date = Column(Date)
   circuit_id = Column(Integer, ForeignKey("circuits.id"))
   circuit = relationship("Circuit", back_populates="races")
   results = relationship("RaceResult", back_populates="race")

class RaceResult(Base):
   __tablename__ = "race_results"
```

```
    id = Column(Integer, primary_key=True, index=True)
    position = Column(Integer)
    points = Column(Float)
    driver_id = Column(Integer, ForeignKey("drivers.id"))
    race_id = Column(Integer, ForeignKey("races.id"))
    driver = relationship("Driver", back_populates="race_results")
    race = relationship("Race", back_populates="results")

# Создание таблиц
Base.metadata.create_all(bind=engine)

# Pydantic модели для валидации данных
class TeamBase(BaseModel):
    name: str
    country: str
    founded_year: int

    class Config:
        from_attributes = True

class DriverBase(BaseModel):
    name: str
    nationality: str
    birth_date: date
    team_id: int

    class Config:
        from_attributes = True

class CarBase(BaseModel):
    model: str
    year: int
    team_id: int

    class Config:
        from_attributes = True

class CircuitBase(BaseModel):
    name: str
    country: str
    length: float

    class Config:
        from_attributes = True

class RaceBase(BaseModel):
    name: str
    date: date
    circuit_id: int
    class Config:
        from_attributes = True

class RaceResultBase(BaseModel):
    position: int
    points: float
    driver_id: int
    race_id: int

    class Config:
        from_attributes = True

# Функция для получения сессии БД
def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()

# Эндпоинты для команд
@app.post("/teams/", response_model=TeamBase)
def create_team(team: TeamBase, db: Session = Depends(get_db)):
    # Проверка, существует ли команда с таким именем
    existing_team = db.query(Team).filter(Team.name == team.name).first()
    if existing_team:
        raise HTTPException(status_code=400, detail="Команда с таким названием уже существует")

    db_team = Team(**team.model_dump())
    db.add(db_team)
    db.commit()
    db.refresh(db_team)
    return db_team

@app.get("/teams/", response_model=List[TeamBase])
def read_teams(skip: int = 0, limit: int = 100, db: Session = Depends(get_db)):
    teams = db.query(Team).offset(skip).limit(limit).all()
    return teams

@app.get("/teams/{team_id}", response_model=TeamBase)
def read_team(team_id: int, db: Session = Depends(get_db)):
    team = db.query(Team).filter(Team.id == team_id).first()
    if team is None:
        raise HTTPException(status_code=404, detail="Team not found")
    return team

@app.put("/teams/{team_id}", response_model=TeamBase)
def update_team(team_id: int, team: TeamBase, db: Session = Depends(get_db)):
    db_team = db.query(Team).filter(Team.id == team_id).first()
    if db_team is None:
        raise HTTPException(status_code=404, detail="Team not found")
    for key, value in team.model_dump().items():
        setattr(db_team, key, value)

    db.commit()
    db.refresh(db_team)
    return db_team

@app.delete("/teams/{team_id}")
def delete_team(team_id: int, db: Session = Depends(get_db)):
    team = db.query(Team).filter(Team.id == team_id).first()
    if team is None:
        raise HTTPException(status_code=404, detail="Team not found")

    db.delete(team)
    db.commit()
    return {"message": "Team deleted successfully"}

# Эндпоинты для гонщиков
@app.post("/drivers/", response_model=DriverBase)
def create_driver(driver: DriverBase, db: Session = Depends(get_db)):
    db_driver = Driver(**driver.model_dump())
    db.add(db_driver)
    db.commit()
    db.refresh(db_driver)
    return db_driver

@app.get("/drivers/", response_model=List[DriverBase])
def read_drivers(skip: int = 0, limit: int = 100, db: Session = Depends(get_db)):
    drivers = db.query(Driver).offset(skip).limit(limit).all()
    return drivers

@app.get("/drivers/{driver_id}", response_model=DriverBase)
def read_driver(driver_id: int, db: Session = Depends(get_db)):
    driver = db.query(Driver).filter(Driver.id == driver_id).first()
    if driver is None:
        raise HTTPException(status_code=404, detail="Driver not found")
    return driver

@app.put("/drivers/{driver_id}", response_model=DriverBase)
def update_driver(driver_id: int, driver: DriverBase, db: Session = Depends(get_db)):
    db_driver = db.query(Driver).filter(Driver.id == driver_id).first()
    if db_driver is None:
        raise HTTPException(status_code=404, detail="Driver not found")

    for key, value in driver.model_dump().items():
        setattr(db_driver, key, value)

    db.commit()
    db.refresh(db_driver)
    return db_driver
```

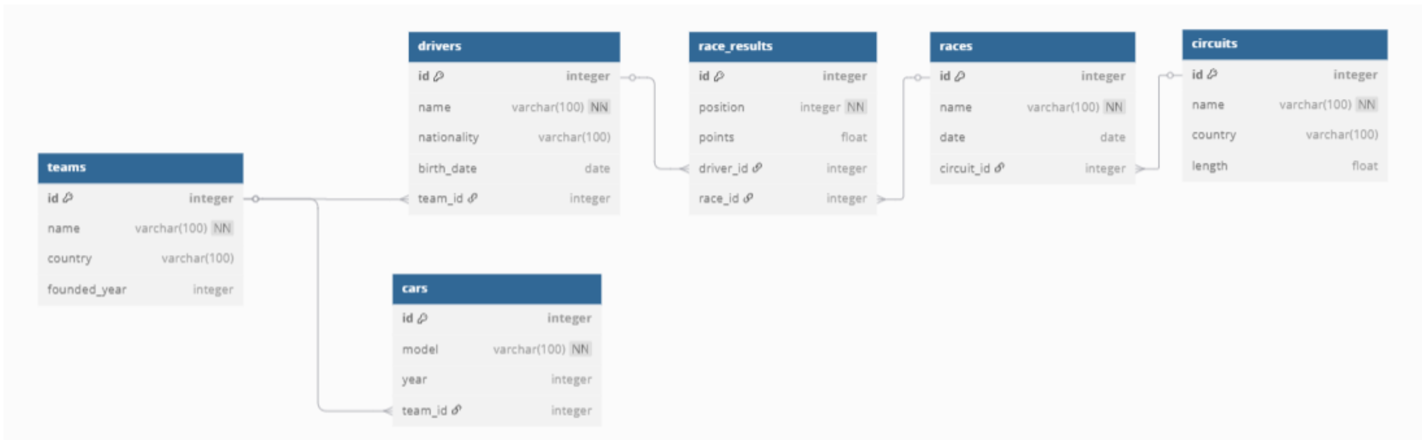
```
@app.delete("/drivers/{driver_id}")
def delete_driver(driver_id: int, db: Session = Depends(get_db)):
    driver = db.query(Driver).filter(Driver.id == driver_id).first()
    if driver is None:
        raise HTTPException(status_code=404, detail="Driver not found")

    db.delete(driver)
    db.commit()
    return {"message": "Driver deleted successfully"}

# Эндпоинты для автомобилей
@app.post("/cars/", response_model=CarBase)
def create_car(car: CarBase, db: Session = Depends(get_db)):
    db_car = Car(**car.model_dump())
    db.add(db_car)
    db.commit()
    db.refresh(db_car)
    return db_car

@app.get("/cars/", response_model=List[CarBase])
def read_cars(skip: int = 0, limit: int = 100, db: Session = Depends(get_db)):
    cars = db.query(Car).offset(skip).limit(limit).all()
    return cars

@app.get("/cars/{car_id}", response_model=CarBase)
def read_car(car_id: int, db: Session = Depends(get_db)):
    car = db.query(Car).filter(Car.id == car_id).first()
    if car is None:
        raise HTTPException(status_code=404, detail="Car not found")
    return car

@app.put("/cars/{car_id}", response_model=CarBase)
def update_car(car_id: int, car: CarBase, db: Session = Depends(get_db)):
    db_car = db.query(Car).filter(Car.id == car_id).first()
    if db_car is None:
        raise HTTPException(status_code=404, detail="Car not found")

    for key, value in car.model_dump().items():
        setattr(db_car, key, value)

    db.commit()
    db.refresh(db_car)
    return db_car

@app.delete("/cars/{car_id}")
def delete_car(car_id: int, db: Session = Depends(get_db)):
    car = db.query(Car).filter(Car.id == car_id).first()
    if car is None:
        raise HTTPException(status_code=404, detail="Car not found")
    db.delete(car)
    db.commit()
    return {"message": "Car deleted successfully"}

# Эндпоинты для трасс
@app.post("/circuits/", response_model=CircuitBase)
def create_circuit(circuit: CircuitBase, db: Session = Depends(get_db)):
    db_circuit = Circuit(**circuit.model_dump())
    db.add(db_circuit)
    db.commit()
    db.refresh(db_circuit)
    return db_circuit

@app.get("/circuits/", response_model=List[CircuitBase])
def read_circuits(skip: int = 0, limit: int = 100, db: Session = Depends(get_db)):
    circuits = db.query(Circuit).offset(skip).limit(limit).all()
    return circuits

@app.get("/circuits/{circuit_id}", response_model=CircuitBase)
def read_circuit(circuit_id: int, db: Session = Depends(get_db)):
    circuit = db.query(Circuit).filter(Circuit.id == circuit_id).first()
    if circuit is None:
        raise HTTPException(status_code=404, detail="Circuit not found")
    return circuit

@app.put("/circuits/{circuit_id}", response_model=CircuitBase)
def update_circuit(circuit_id: int, circuit: CircuitBase, db: Session = Depends(get_db)):
    db_circuit = db.query(Circuit).filter(Circuit.id == circuit_id).first()
    if db_circuit is None:
        raise HTTPException(status_code=404, detail="Circuit not found")

    for key, value in circuit.model_dump().items():
        setattr(db_circuit, key, value)

    db.commit()
    db.refresh(db_circuit)
    return db_circuit

@app.delete("/circuits/{circuit_id}")
def delete_circuit(circuit_id: int, db: Session = Depends(get_db)):
    circuit = db.query(Circuit).filter(Circuit.id == circuit_id).first()
    if circuit is None:
        raise HTTPException(status_code=404, detail="Circuit not found")

    db.delete(circuit)
    db.commit()
    return {"message": "Circuit deleted successfully"}

# Эндпоинты для гонок
@app.post("/races/", response_model=RaceBase)
def create_race(race: RaceBase, db: Session = Depends(get_db)):
    db_race = Race(**race.model_dump())
    db.add(db_race)
    db.commit()
    db.refresh(db_race)
    return db_race

@app.get("/races/", response_model=List[RaceBase])
def read_races(skip: int = 0, limit: int = 100, db: Session = Depends(get_db)):
    races = db.query(Race).offset(skip).limit(limit).all()
    return races

@app.get("/races/{race_id}", response_model=RaceBase)
def read_race(race_id: int, db: Session = Depends(get_db)):
    race = db.query(Race).filter(Race.id == race_id).first()
    if race is None:
        raise HTTPException(status_code=404, detail="Race not found")
    return race

@app.put("/races/{race_id}", response_model=RaceBase)
def update_race(race_id: int, race: RaceBase, db: Session = Depends(get_db)):
    db_race = db.query(Race).filter(Race.id == race_id).first()
    if db_race is None:
        raise HTTPException(status_code=404, detail="Race not found")

    for key, value in race.model_dump().items():
        setattr(db_race, key, value)

    db.commit()
    db.refresh(db_race)
    return db_race

@app.delete("/races/{race_id}")
def delete_race(race_id: int, db: Session = Depends(get_db)):
    race = db.query(Race).filter(Race.id == race_id).first()
    if race is None:
        raise HTTPException(status_code=404, detail="Race not found")

    db.delete(race)
    db.commit()
    return {"message": "Race deleted successfully"}

# Эндпоинты для результатов гонок
@app.post("/race_results/", response_model=RaceResultBase)
def create_race_result(race_result: RaceResultBase, db: Session = Depends(get_db)):
    db_race_result = RaceResult(**race_result.model_dump())
    db.add(db_race_result)
    db.commit()
    db.refresh(db_race_result)
    return db_race_result

@app.get("/race_results/", response_model=List[RaceResultBase])
```

```
 def read_race_results(skip: int = 0, limit: int = 100, db: Session = Depends(get_db)):
     race_results = db.query(RaceResult).offset(skip).limit(limit).all()
     return race_results

 @app.get("/race_results/{race_result_id}", response_model=RaceResultBase)
 def read_race_result(race_result_id: int, db: Session = Depends(get_db)):
     race_result = db.query(RaceResult).filter(RaceResult.id == race_result_id).first()
     if race_result is None:
         raise HTTPException(status_code=404, detail="Race result not found")
     return race_result

 @app.put("/race_results/{race_result_id}", response_model=RaceResultBase)
 def update_race_result(race_result_id: int, race_result: RaceResultBase, db: Session =
Depends(get_db)):
     db_race_result = db.query(RaceResult).filter(RaceResult.id == race_result_id).first()
     if db_race_result is None:
         raise HTTPException(status_code=404, detail="Race result not found")

     for key, value in race_result.model_dump().items():
         setattr(db_race_result, key, value)

     db.commit()
     db.refresh(db_race_result)
     return db_race_result

 @app.delete("/race_results/{race_result_id}")
 def delete_race_result(race_result_id: int, db: Session = Depends(get_db)):
     race_result = db.query(RaceResult).filter(RaceResult.id == race_result_id).first()
     if race_result is None:
         raise HTTPException(status_code=404, detail="Race result not found")

     db.delete(race_result)
     db.commit()
     return {"message": "Race result deleted successfully"}

 if __name__ == "__main__":
     import uvicorn
     uvicorn.run(app, host="0.0.0.0", port=8000)
```

Рисунки с результатами работы программы:



```
(venv) PS C:\Users\Nikita\Documents\GitHub\spp_po11\reports\Antonyuk\5\src> python SPP_Lab5_Task1.py
C:\Users\Nikita\Documents\GitHub\spp_po11\reports\Antonyuk\5\src\SPP_Lab5_Task1.py:26: MovedIn20Warning: The ``declarative_ba
se()`` function is now available as sqlalchemy.orm.declarative_base(). (deprecated since: 2.0) (Background on SQLAlchemy 2.0
at: https://sqlalche.me/e/b8d9)
  Base = declarative_base()
INFO:     Started server process [8912]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
```



Вывод: приобрел практические навыки разработки API и баз данных.