Министерство образования Республики Беларусь
Учреждение образования
«Брестский государственный технический университет»
Кафедра ИИТ

Лабораторная работа №5
По дисциплине:“ Современные платформы программирования”

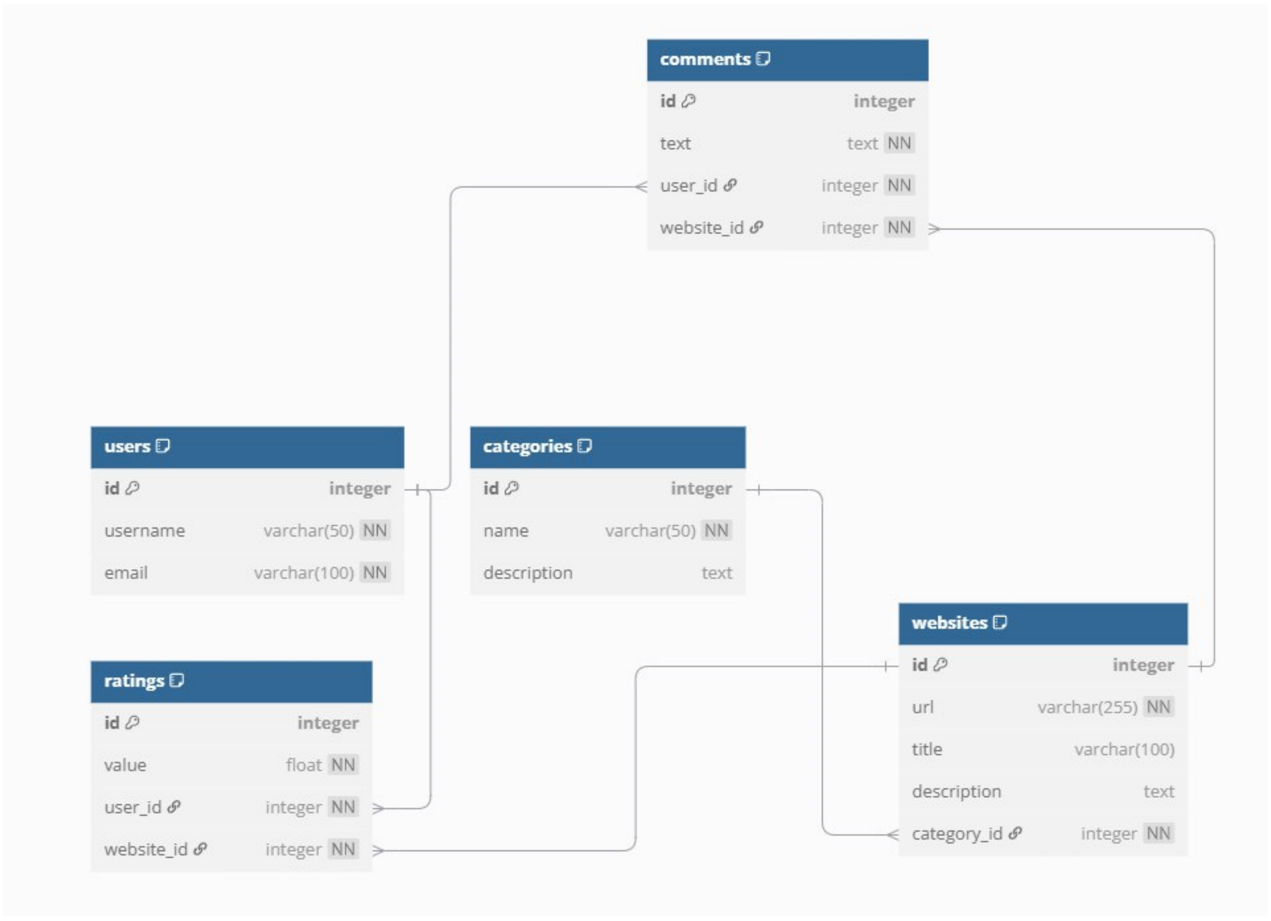
Выполнил:
Студент 3 курса
Группы ПО-11
Сологуб А.В.
Проверил:
Козик И.Д.

Брест 2025

Цель: приобрести практические навыки разработки API и баз данных
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;

17) База данных Ресурсы Internet

Структура БД:
```
from sqlalchemy import Column, Integer, String, Float, ForeignKey, Text
from sqlalchemy.orm import relationship
from database import Base

class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True, index=True)
    username = Column(String(50), unique=True, nullable=False)
    email = Column(String(100), unique=True, nullable=False)
    comments = relationship("Comment", back_populates="author")
    ratings = relationship("Rating", back_populates="user")

class Category(Base):
    __tablename__ = 'categories'
    id = Column(Integer, primary_key=True, index=True)
    name = Column(String(50), unique=True, nullable=False)
    description = Column(Text, nullable=True)
    websites = relationship("Website", back_populates="category")

class Website(Base):
    __tablename__ = 'websites'
    id = Column(Integer, primary_key=True, index=True)
    url = Column(String(255), unique=True, nullable=False)
    title = Column(String(100), nullable=True)
    description = Column(Text, nullable=True)
    category_id = Column(Integer, ForeignKey('categories.id'))
    category = relationship("Category", back_populates="websites")
    comments = relationship("Comment", back_populates="website")
    ratings = relationship("Rating", back_populates="website")

class Comment(Base):
    __tablename__ = 'comments'
    id = Column(Integer, primary_key=True, index=True)
    text = Column(Text, nullable=False)
    user_id = Column(Integer, ForeignKey('users.id'))
    website_id = Column(Integer, ForeignKey('websites.id'))
    author = relationship("User", back_populates="comments")
    website = relationship("Website", back_populates="comments")

class Rating(Base):
    __tablename__ = 'ratings'
    id = Column(Integer, primary_key=True, index=True)
    value = Column(Float, nullable=False)
    user_id = Column(Integer, ForeignKey('users.id'))
    website_id = Column(Integer, ForeignKey('websites.id'))
    user = relationship("User", back_populates="ratings")
    website = relationship("Website", back_populates="ratings")
```



Database.py:

```
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

SQLALCHEMY_DATABASE_URL = "sqlite:///./internet_resources.db"

engine = create_engine(
    SQLALCHEMY_DATABASE_URL,
    connect_args={"check_same_thread": False}
)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

Base = declarative_base()
```

models.py:

```
from pydantic import BaseModel
from typing import Optional

class UserBase(BaseModel):
    username: str
    email: str

class UserCreate(UserBase):
    pass

class User(UserBase):
    id: int
    class Config:
        from_attributes = True

class CategoryBase(BaseModel):
    name: str
    description: Optional[str] = None

class CategoryCreate(CategoryBase):
    pass

class Category(CategoryBase):
    id: int
    class Config:
        from_attributes = True

class WebsiteBase(BaseModel):
    url: str
    title: str
    description: Optional[str] = None
    category_id: int

class WebsiteCreate(WebsiteBase):
    pass

class Website(WebsiteBase):
    id: int
    class Config:
        from_attributes = True

class CommentBase(BaseModel):
    text: str
    user_id: int
    website_id: int

class CommentCreate(CommentBase):
    pass

class Comment(CommentBase):
    id: int
    class Config:
        from_attributes = True

class RatingBase(BaseModel):
    value: float
    user_id: int
    website_id: int

class RatingCreate(RatingBase):
    pass

class Rating(RatingBase):
    id: int
    class Config:
        from_attributes = True
```

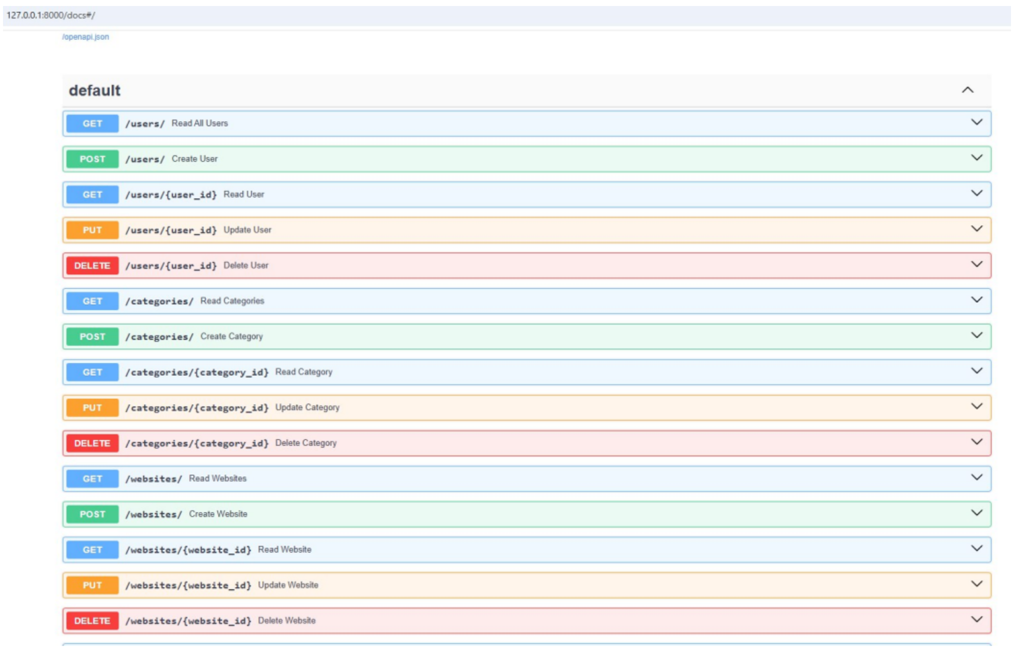
CRUD для каждой таблицы

```
from fastapi import FastAPI, Depends, HTTPException
from sqlalchemy.orm import Session
from typing import List
import models
import schemas
from database import SessionLocal, engine

# Создаем таблицы
models.Base.metadata.create_all(bind=engine)

app = FastAPI()
```

```
# Dependency
def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()

@app.get("/users/", response_model=List[schemas.User])
def read_all_users(
    skip: int = 0,
    limit: int = 100,
    db: Session = Depends(get_db)
):
    users = db.query(models.User).offset(skip).limit(limit).all()
    return users

@app.post("/users/", response_model=schemas.User)
def create_user(user: schemas.UserCreate, db: Session = Depends(get_db)):
    try:
        # Проверка уникальности
        if db.query(models.User).filter(models.User.email == user.email).first():
            raise HTTPException(status_code=400, detail="Email already registered")
        if db.query(models.User).filter(models.User.username == user.username).first():
            raise HTTPException(status_code=400, detail="Username already taken")

        db_user = models.User(**user.dict())
        db.add(db_user)
        db.commit()
        db.refresh(db_user)
        return db_user
    except Exception as e:
        db.rollback()
        raise HTTPException(status_code=500, detail=str(e))


# GET /users/{user_id} - получение одного пользователя
@app.get("/users/{user_id}", response_model=schemas.User)
def read_user(user_id: int, db: Session = Depends(get_db)):
    db_user = db.query(models.User).filter(models.User.id == user_id).first()
    if not db_user:
        raise HTTPException(status_code=404, detail="User not found")
    return db_user


# PUT /users/{user_id} - обновление пользователя
@app.put("/users/{user_id}", response_model=schemas.User)
def update_user(user_id: int, user: schemas.UserCreate, db: Session = Depends(get_db)):
    db_user = db.query(models.User).filter(models.User.id == user_id).first()
    if not db_user:
        raise HTTPException(status_code=404, detail="User not found")

    # Проверка уникальности новых данных
    if user.email != db_user.email and db.query(models.User).filter(models.User.email ==
user.email).first():
        raise HTTPException(status_code=400, detail="Email already registered")
    if user.username != db_user.username and db.query(models.User).filter(
            models.User.username == user.username).first():
        raise HTTPException(status_code=400, detail="Username already taken")

    for field, value in user.dict().items():
        setattr(db_user, field, value)

    db.commit()
    db.refresh(db_user)
    return db_user


# DELETE /users/{user_id} - удаление пользователя
@app.delete("/users/{user_id}")
def delete_user(user_id: int, db: Session = Depends(get_db)):
    db_user = db.query(models.User).filter(models.User.id == user_id).first()
    if not db_user:
        raise HTTPException(status_code=404, detail="User not found")

    db.delete(db_user)
    db.commit()
    return {"message": "User deleted successfully"}

@app.get("/categories/", response_model=List[schemas.Category])
def read_categories(skip: int = 0, limit: int = 100, db: Session = Depends(get_db)):
    return db.query(models.Category).offset(skip).limit(limit).all()

@app.post("/categories/", response_model=schemas.Category)
def create_category(category: schemas.CategoryCreate, db: Session = Depends(get_db)):
    try:
        if db.query(models.Category).filter(models.Category.name == category.name).first():
            raise HTTPException(status_code=400, detail="Category already exists")

        db_category = models.Category(**category.dict())
        db.add(db_category)
        db.commit()
        db.refresh(db_category)
        return db_category
    except Exception as e:
        db.rollback()
        raise HTTPException(status_code=500, detail=str(e))


# GET /categories/{category_id}
@app.get("/categories/{category_id}", response_model=schemas.Category)
def read_category(category_id: int, db: Session = Depends(get_db)):
    db_category = db.query(models.Category).filter(models.Category.id == category_id).first()
    if not db_category:
        raise HTTPException(status_code=404, detail="Category not found")
    return db_category


# PUT /categories/{category_id}
@app.put("/categories/{category_id}", response_model=schemas.Category)
def update_category(category_id: int, category: schemas.CategoryCreate, db: Session = Depends(get_db)):
    db_category = db.query(models.Category).filter(models.Category.id == category_id).first()
    if not db_category:
        raise HTTPException(status_code=404, detail="Category not found")

    if category.name != db_category.name and db.query(models.Category).filter(
            models.Category.name == category.name).first():
        raise HTTPException(status_code=400, detail="Category name already exists")

    for field, value in category.dict().items():
        setattr(db_category, field, value)

    db.commit()
    db.refresh(db_category)
    return db_category


# DELETE /categories/{category_id}
@app.delete("/categories/{category_id}")
def delete_category(category_id: int, db: Session = Depends(get_db)):
    db_category = db.query(models.Category).filter(models.Category.id == category_id).first()
    if not db_category:
        raise HTTPException(status_code=404, detail="Category not found")

    # Проверка связанных сайтов
    if db.query(models.Website).filter(models.Website.category_id == category_id).first():
        raise HTTPException(status_code=400, detail="Cannot delete category with associated websites")

    db.delete(db_category)
    db.commit()
    return {"message": "Category deleted successfully"}

@app.post("/websites/", response_model=schemas.Website)
def create_website(website: schemas.WebsiteCreate, db: Session = Depends(get_db)):
    try:
        if not db.query(models.Category).filter(models.Category.id == website.category_id).first():
            raise HTTPException(status_code=404, detail="Category not found")
        if db.query(models.Website).filter(models.Website.url == website.url).first():
            raise HTTPException(status_code=400, detail="Website URL already exists")

        db_website = models.Website(**website.dict())
        db.add(db_website)
        db.commit()
        db.refresh(db_website)
        return db_website
    except Exception as e:
        db.rollback()
        raise HTTPException(status_code=500, detail=str(e))


# GET /websites/{website_id}
@app.get("/websites/{website_id}", response_model=schemas.Website)
def read_website(website_id: int, db: Session = Depends(get_db)):
    db_website = db.query(models.Website).filter(models.Website.id == website_id).first()
    if not db_website:
        raise HTTPException(status_code=404, detail="Website not found")
    return db_website


# PUT /websites/{website_id}
@app.put("/websites/{website_id}", response_model=schemas.Website)
def update_website(website_id: int, website: schemas.WebsiteCreate, db: Session = Depends(get_db)):
    db_website = db.query(models.Website).filter(models.Website.id == website_id).first()
    if not db_website:
        raise HTTPException(status_code=404, detail="Website not found")

    if not db.query(models.Category).filter(models.Category.id == website.category_id).first():
        raise HTTPException(status_code=404, detail="Category not found")

    if website.url != db_website.url and db.query(models.Website).filter(models.Website.url ==
website.url).first():
        raise HTTPException(status_code=400, detail="Website URL already exists")

    for field, value in website.dict().items():
        setattr(db_website, field, value)

    db.commit()
```

```python
    db.refresh(db_website)
    return db_website


# DELETE /websites/{website_id}
@app.delete("/websites/{website_id}")
def delete_website(website_id: int, db: Session = Depends(get_db)):
    db_website = db.query(models.Website).filter(models.Website.id == website_id).first()
    if not db_website:
        raise HTTPException(status_code=404, detail="Website not found")

    db.delete(db_website)
    db.commit()
    return {"message": "Website deleted successfully"}

@app.post("/comments/", response_model=schemas.Comment)
def create_comment(comment: schemas.CommentCreate, db: Session = Depends(get_db)):
    try:
        if not db.query(models.User).filter(models.User.id == comment.user_id).first():
            raise HTTPException(status_code=404, detail="User not found")
        if not db.query(models.Website).filter(models.Website.id == comment.website_id).first():
            raise HTTPException(status_code=404, detail="Website not found")

        db_comment = models.Comment(**comment.dict())
        db.add(db_comment)
        db.commit()
        db.refresh(db_comment)
        return db_comment
    except Exception as e:
        db.rollback()
        raise HTTPException(status_code=500, detail=str(e))


# GET /comments/{comment_id}
@app.get("/comments/{comment_id}", response_model=schemas.Comment)
def read_comment(comment_id: int, db: Session = Depends(get_db)):
    db_comment = db.query(models.Comment).filter(models.Comment.id == comment_id).first()
    if not db_comment:
        raise HTTPException(status_code=404, detail="Comment not found")
    return db_comment


# PUT /comments/{comment_id}
@app.put("/comments/{comment_id}", response_model=schemas.Comment)
def update_comment(comment_id: int, comment: schemas.CommentCreate, db: Session = Depends(get_db)):
    db_comment = db.query(models.Comment).filter(models.Comment.id == comment_id).first()
    if not db_comment:
        raise HTTPException(status_code=404, detail="Comment not found")

    if not db.query(models.User).filter(models.User.id == comment.user_id).first():
        raise HTTPException(status_code=404, detail="User not found")
    if not db.query(models.Website).filter(models.Website.id == comment.website_id).first():
        raise HTTPException(status_code=404, detail="Website not found")

    for field, value in comment.dict().items():
        setattr(db_comment, field, value)

    db.commit()
    db.refresh(db_comment)
    return db_comment


# DELETE /comments/{comment_id}
```

```
@app.delete("/comments/{comment_id}")
def delete_comment(comment_id: int, db: Session = Depends(get_db)):
    db_comment = db.query(models.Comment).filter(models.Comment.id == comment_id).first()
    if not db_comment:
        raise HTTPException(status_code=404, detail="Comment not found")

    db.delete(db_comment)
    db.commit()
    return {"message": "Comment deleted successfully"}

@app.post("/ratings/", response_model=schemas.Rating)
def create_rating(rating: schemas.RatingCreate, db: Session = Depends(get_db)):
    try:
        if not db.query(models.User).filter(models.User.id == rating.user_id).first():
            raise HTTPException(status_code=404, detail="User not found")
        if not db.query(models.Website).filter(models.Website.id == rating.website_id).first():
            raise HTTPException(status_code=404, detail="Website not found")

        db_rating = models.Rating(**rating.dict())
        db.add(db_rating)
        db.commit()
        db.refresh(db_rating)
        return db_rating
    except Exception as e:
        db.rollback()
        raise HTTPException(status_code=500, detail=str(e))

    # GET /ratings/{rating_id}
    @app.get("/ratings/{rating_id}", response_model=schemas.Rating)
    def read_rating(rating_id: int, db: Session = Depends(get_db)):
        db_rating = db.query(models.Rating).filter(models.Rating.id == rating_id).first()
        if not db_rating:
            raise HTTPException(status_code=404, detail="Rating not found")
        return db_rating

    # PUT /ratings/{rating_id}
    @app.put("/ratings/{rating_id}", response_model=schemas.Rating)
    def update_rating(rating_id: int, rating: schemas.RatingCreate, db: Session = Depends(get_db)):
        db_rating = db.query(models.Rating).filter(models.Rating.id == rating_id).first()
        if not db_rating:
            raise HTTPException(status_code=404, detail="Rating not found")

        if not db.query(models.User).filter(models.User.id == rating.user_id).first():
            raise HTTPException(status_code=404, detail="User not found")
        if not db.query(models.Website).filter(models.Website.id == rating.website_id).first():
            raise HTTPException(status_code=404, detail="Website not found")

        for field, value in rating.dict().items():
            setattr(db_rating, field, value)

        db.commit()
        db.refresh(db_rating)
        return db_rating

    # DELETE /ratings/{rating_id}
    @app.delete("/ratings/{rating_id}")
    def delete_rating(rating_id: int, db: Session = Depends(get_db)):
        db_rating = db.query(models.Rating).filter(models.Rating.id == rating_id).first()
        if not db_rating:
            raise HTTPException(status_code=404, detail="Rating not found")

        db.delete(db_rating)
        db.commit()
        return {"message": "Rating deleted successfully"}

@app.get("/")
def read_root():
    return {"message": "Internet Resources API"}
```

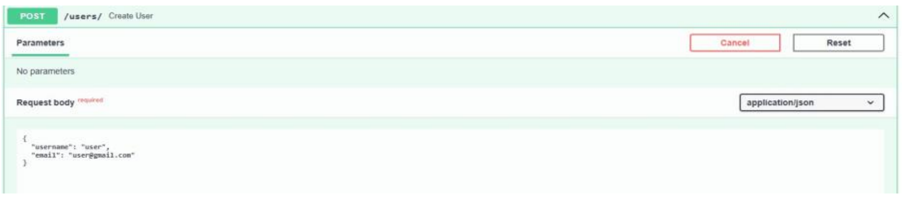


127.0.0.1:8000/docs#/

/openapi.json

default

GET	/users/	Read All Users
POST	/users/	Create User
GET	/users/{user_id}	Read User
PUT	/users/{user_id}	Update User
DELETE	/users/{user_id}	Delete User
GET	/categories/	Read Categories
POST	/categories/	Create Category
GET	/categories/{category_id}	Read Category
PUT	/categories/{category_id}	Update Category
DELETE	/categories/{category_id}	Delete Category
GET	/websites/	Read Websites
POST	/websites/	Create Website
GET	/websites/{website_id}	Read Website
PUT	/websites/{website_id}	Update Website
DELETE	/websites/{website_id}	Delete Website



POST /users/ Create User

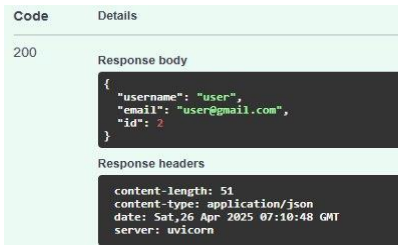
Parameters

No parameters

Request body required

application/json

```
{
  "username": "user",
  "email": "user@gmail.com"
}
```



Code	Details
200	Response body

```
{
  "username": "user",
  "email": "user@gmail.com",
  "id": 2
}
```

Response headers

```
 content-length: 51
 content-type: application/json
 date: Sat,26 Apr 2025 07:10:48 GMT
 server: uvicorn
```

Вывод: приобрела практические навыки разработки API и баз данных