МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ

«БРЕСТСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

ФАКУЛЬТЕТ ЭЛЕКТРОННО-ИНФОРМАЦИОННЫХ СИСТЕМ

Кафедра интеллектуальных информационных технологий

Отчёт по лабораторной работе №5

Специальность ПО

Выполнил

В. С. Юрашевич,

студент группы ПО-11

Проверил

И. Д. Козик

Брест 2025

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

Ход работы

База данных: Бухгалтерия.

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;
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;

Код программы:

```
from fastapi import FastAPI, HTTPException

from pydantic import BaseModel

from typing import List, Optional

from datetime import date, datetime

from sqlalchemy import create_engine, Column, Integer, String, Float, Date, DateTime, ForeignKey, Boolean, Enum

from sqlalchemy.orm import declarative_base, relationship, sessionmaker

import enum


app = FastAPI()


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

engine = create_engine(SQLALCHEMY_DATABASE_URL)

SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)


Base = declarative_base()
```

```
class AccountType(enum.Enum):
    asset = "asset"
    liability = "liability"
    equity = "equity"
    revenue = "revenue"
    expense = "expense"


class TransactionType(enum.Enum):
    debit = "debit"
    credit = "credit"


class InvoiceStatus(enum.Enum):
    draft = "draft"
    sent = "sent"
    paid = "paid"
    overdue = "overdue"
    cancelled = "cancelled"

class Organization(Base):
    __tablename__ = "organizations"

    id = Column(Integer, primary_key=True, index=True)
    name = Column(String, nullable=False)
    tax_id = Column(String, unique=True)
    address = Column(String)
    phone = Column(String)
    email = Column(String)
```

```
    accounts = relationship("Account", back_populates="organization")
    invoices = relationship("Invoice", back_populates="organization")


class Account(Base):
    __tablename__ = "accounts"

    id = Column(Integer, primary_key=True, index=True)
    name = Column(String, nullable=False)
    code = Column(String, unique=True, nullable=False)
    account_type = Column(Enum(AccountType), nullable=False)
    balance = Column(Float, default=0.0)
    organization_id = Column(Integer, ForeignKey("organizations.id"))

    organization = relationship("Organization", back_populates="accounts")
    transactions = relationship("Transaction", back_populates="account")


class Client(Base):
    __tablename__ = "clients"

    id = Column(Integer, primary_key=True, index=True)
    name = Column(String, nullable=False)
    tax_id = Column(String, unique=True)
    address = Column(String)
    phone = Column(String)
    email = Column(String)

    invoices = relationship("Invoice", back_populates="client")
```

```
class Invoice(Base):
    __tablename__ = "invoices"

    id = Column(Integer, primary_key=True, index=True)
    number = Column(String, unique=True, nullable=False)
    issue_date = Column(Date, default=date.today())
    due_date = Column(Date)
    amount = Column(Float, nullable=False)
    tax_amount = Column(Float, default=0.0)
    total_amount = Column(Float, nullable=False)
    status = Column(Enum(InvoiceStatus), default=InvoiceStatus.draft)
    client_id = Column(Integer, ForeignKey("clients.id"))
    organization_id = Column(Integer, ForeignKey("organizations.id"))

    client = relationship("Client", back_populates="invoices")
    organization = relationship("Organization", back_populates="invoices")
    transactions = relationship("Transaction", back_populates="invoice")


class Transaction(Base):
    __tablename__ = "transactions"

    id = Column(Integer, primary_key=True, index=True)
    date = Column(DateTime, default=datetime.now())
    amount = Column(Float, nullable=False)
    description = Column(String)
    transaction_type = Column(Enum(TransactionType), nullable=False)
    account_id = Column(Integer, ForeignKey("accounts.id"))
    invoice_id = Column(Integer, ForeignKey("invoices.id"), nullable=True)

    account = relationship("Account", back_populates="transactions")
```

```
    invoice = relationship("Invoice", back_populates="transactions")


Base.metadata.create_all(bind=engine)


class OrganizationCreate(BaseModel):
    name: str
    tax_id: str
    address: Optional[str] = None
    phone: Optional[str] = None
    email: Optional[str] = None



class OrganizationResponse(BaseModel):
    id: int
    name: str
    tax_id: str
    address: Optional[str] = None
    phone: Optional[str] = None
    email: Optional[str] = None

    class Config:
        from_attributes = True



class AccountCreate(BaseModel):
    name: str
    code: str
    account_type: AccountType
    organization_id: int
```

```
class AccountResponse(BaseModel):
    id: int
    name: str
    code: str
    account_type: AccountType
    balance: float
    organization_id: int

    class Config:
        from_attributes = True


class ClientCreate(BaseModel):
    name: str
    tax_id: str
    address: Optional[str] = None
    phone: Optional[str] = None
    email: Optional[str] = None


class ClientResponse(BaseModel):
    id: int
    name: str
    tax_id: str
    address: Optional[str] = None
    phone: Optional[str] = None
    email: Optional[str] = None

    class Config:
        from_attributes = True
```

```
class InvoiceCreate(BaseModel):
    number: str
    issue_date: date = date.today()
    due_date: date
    amount: float
    tax_amount: float = 0.0
    client_id: int
    organization_id: int



class InvoiceResponse(BaseModel):
    id: int
    number: str
    issue_date: date
    due_date: date
    amount: float
    tax_amount: float
    total_amount: float
    status: InvoiceStatus
    client_id: int
    organization_id: int

    class Config:
        from_attributes = True



class TransactionCreate(BaseModel):
    amount: float
    description: Optional[str] = None
    transaction_type: TransactionType
```

```
    account_id: int
    invoice_id: Optional[int] = None


class TransactionResponse(BaseModel):
    id: int
    date: datetime
    amount: float
    description: Optional[str] = None
    transaction_type: TransactionType
    account_id: int
    invoice_id: Optional[int] = None

    class Config:
        from_attributes = True


@app.post("/organizations/", response_model=OrganizationResponse)
def create_organization(organization: OrganizationCreate):
    db = SessionLocal()
    db_org = Organization(**organization.dict())
    db.add(db_org)
    db.commit()
    db.refresh(db_org)
    db.close()
    return db_org


@app.get("/organizations/", response_model=List[OrganizationResponse])
def read_organizations():
    db = SessionLocal()
```

```
    orgs = db.query(Organization).all()
    db.close()
    return orgs


@app.get("/organizations/{org_id}", response_model=OrganizationResponse)
def read_organization(org_id: int):
    db = SessionLocal()
    org = db.query(Organization).filter(Organization.id == org_id).first()
    db.close()
    if not org:
        raise HTTPException(status_code=404, detail="Organization not found")
    return org


@app.put("/organizations/{org_id}", response_model=OrganizationResponse)
def update_organization(org_id: int, organization: OrganizationCreate):
    db = SessionLocal()
    db_org = db.query(Organization).filter(Organization.id == org_id).first()
    if not db_org:
        db.close()
        raise HTTPException(status_code=404, detail="Organization not found")

    for key, value in organization.dict().items():
        setattr(db_org, key, value)

    db.commit()
    db.refresh(db_org)
    db.close()
    return db_org
```

```
@app.delete("/organizations/{org_id}")
def delete_organization(org_id: int):
    db = SessionLocal()
    org = db.query(Organization).filter(Organization.id == org_id).first()
    if not org:
        db.close()
        raise HTTPException(status_code=404, detail="Organization not found")

    db.delete(org)
    db.commit()
    db.close()
    return {"message": "Organization deleted"}

@app.post("/accounts/", response_model=AccountResponse)
def create_account(account: AccountCreate):
    db = SessionLocal()

    org = db.query(Organization).filter(Organization.id ==
account.organization_id).first()
    if not org:
        db.close()
        raise HTTPException(status_code=404, detail="Organization not found")

    db_account = Account(**account.dict())
    db.add(db_account)
    db.commit()
    db.refresh(db_account)
    db.close()
    return db_account
```

```
@app.get("/accounts/", response_model=List[AccountResponse])
def read_accounts():
    db = SessionLocal()
    accounts = db.query(Account).all()
    db.close()
    return accounts


@app.get("/accounts/{account_id}", response_model=AccountResponse)
def read_account(account_id: int):
    db = SessionLocal()
    account = db.query(Account).filter(Account.id == account_id).first()
    db.close()
    if not account:
        raise HTTPException(status_code=404, detail="Account not found")
    return account


@app.put("/accounts/{account_id}", response_model=AccountResponse)
def update_account(account_id: int, account: AccountCreate):
    db = SessionLocal()
    db_account = db.query(Account).filter(Account.id == account_id).first()
    if not db_account:
        db.close()
        raise HTTPException(status_code=404, detail="Account not found")

    org = db.query(Organization).filter(Organization.id ==
account.organization_id).first()
    if not org:
        db.close()
```

```
        raise HTTPException(status_code=404, detail="Organization not found")

    for key, value in account.dict().items():
        setattr(db_account, key, value)

    db.commit()
    db.refresh(db_account)
    db.close()
    return db_account



@app.delete("/accounts/{account_id}")
def delete_account(account_id: int):
    db = SessionLocal()
    account = db.query(Account).filter(Account.id == account_id).first()
    if not account:
        db.close()
        raise HTTPException(status_code=404, detail="Account not found")

    db.delete(account)
    db.commit()
    db.close()
    return {"message": "Account deleted"}

@app.post("/clients/", response_model=ClientResponse)
def create_client(client: ClientCreate):
    db = SessionLocal()
    db_client = Client(**client.dict())
    db.add(db_client)
    db.commit()
    db.refresh(db_client)
```

```
    db.close()
    return db_client


@app.get("/clients/", response_model=List[ClientResponse])
def read_clients():
    db = SessionLocal()
    clients = db.query(Client).all()
    db.close()
    return clients


@app.get("/clients/{client_id}", response_model=ClientResponse)
def read_client(client_id: int):
    db = SessionLocal()
    client = db.query(Client).filter(Client.id == client_id).first()
    db.close()
    if not client:
        raise HTTPException(status_code=404, detail="Client not found")
    return client


@app.put("/clients/{client_id}", response_model=ClientResponse)
def update_client(client_id: int, client: ClientCreate):
    db = SessionLocal()
    db_client = db.query(Client).filter(Client.id == client_id).first()
    if not db_client:
        db.close()
        raise HTTPException(status_code=404, detail="Client not found")

    for key, value in client.dict().items():
```

```
            setattr(db_client, key, value)

    db.commit()
    db.refresh(db_client)
    db.close()
    return db_client


@app.delete("/clients/{client_id}")
def delete_client(client_id: int):
    db = SessionLocal()
    client = db.query(Client).filter(Client.id == client_id).first()
    if not client:
        db.close()
        raise HTTPException(status_code=404, detail="Client not found")

    db.delete(client)
    db.commit()
    db.close()
    return {"message": "Client deleted"}

@app.post("/invoices/", response_model=InvoiceResponse)
def create_invoice(invoice: InvoiceCreate):
    db = SessionLocal()

    client = db.query(Client).filter(Client.id == invoice.client_id).first()
    if not client:
        db.close()
        raise HTTPException(status_code=404, detail="Client not found")
```

```
    org = db.query(Organization).filter(Organization.id ==
invoice.organization_id).first()
    if not org:
        db.close()
        raise HTTPException(status_code=404, detail="Organization not found")

    total_amount = invoice.amount + invoice.tax_amount
    db_invoice = Invoice(
        **invoice.dict(),
        total_amount=total_amount,
        status=InvoiceStatus.draft
    )
    db.add(db_invoice)
    db.commit()
    db.refresh(db_invoice)
    db.close()
    return db_invoice



@app.get("/invoices/", response_model=List[InvoiceResponse])
def read_invoices():
    db = SessionLocal()
    invoices = db.query(Invoice).all()
    db.close()
    return invoices



@app.get("/invoices/{invoice_id}", response_model=InvoiceResponse)
def read_invoice(invoice_id: int):
    db = SessionLocal()
    invoice = db.query(Invoice).filter(Invoice.id == invoice_id).first()
```

```
    db.close()
    if not invoice:
        raise HTTPException(status_code=404, detail="Invoice not found")
    return invoice


@app.put("/invoices/{invoice_id}", response_model=InvoiceResponse)
def update_invoice(invoice_id: int, invoice: InvoiceCreate):
    db = SessionLocal()
    db_invoice = db.query(Invoice).filter(Invoice.id == invoice_id).first()
    if not db_invoice:
        db.close()
        raise HTTPException(status_code=404, detail="Invoice not found")

    client = db.query(Client).filter(Client.id == invoice.client_id).first()
    if not client:
        db.close()
        raise HTTPException(status_code=404, detail="Client not found")

    org = db.query(Organization).filter(Organization.id ==
invoice.organization_id).first()
    if not org:
        db.close()
        raise HTTPException(status_code=404, detail="Organization not found")

    total_amount = invoice.amount + invoice.tax_amount

    for key, value in invoice.dict().items():
        setattr(db_invoice, key, value)

    db_invoice.total_amount = total_amount
```

```
    db.commit()
    db.refresh(db_invoice)
    db.close()
    return db_invoice


@app.delete("/invoices/{invoice_id}")
def delete_invoice(invoice_id: int):
    db = SessionLocal()
    invoice = db.query(Invoice).filter(Invoice.id == invoice_id).first()
    if not invoice:
        db.close()
        raise HTTPException(status_code=404, detail="Invoice not found")

    db.delete(invoice)
    db.commit()
    db.close()
    return {"message": "Invoice deleted"}


@app.post("/invoices/{invoice_id}/send")
def send_invoice(invoice_id: int):
    db = SessionLocal()
    invoice = db.query(Invoice).filter(Invoice.id == invoice_id).first()
    if not invoice:
        db.close()
        raise HTTPException(status_code=404, detail="Invoice not found")

    if invoice.status != InvoiceStatus.draft:
        db.close()
        raise HTTPException(status_code=400, detail="Invoice already sent")
```

```
    invoice.status = InvoiceStatus.sent
    db.commit()
    db.close()
    return {"message": "Invoice sent successfully"}


@app.post("/transactions/", response_model=TransactionResponse)
def create_transaction(transaction: TransactionCreate):
    db = SessionLocal()

    account = db.query(Account).filter(Account.id == transaction.account_id).first()
    if not account:
        db.close()
        raise HTTPException(status_code=404, detail="Account not found")

    if transaction.invoice_id is not None:
        invoice = db.query(Invoice).filter(Invoice.id ==
transaction.invoice_id).first()
        if not invoice:
            db.close()
            raise HTTPException(status_code=404, detail="Invoice not found")

    db_transaction = Transaction(**transaction.dict())
    db.add(db_transaction)

    if transaction.transaction_type == TransactionType.debit:
        account.balance += transaction.amount
    else:
        account.balance -= transaction.amount

    db.commit()
```

```
    db.refresh(db_transaction)
    db.close()
    return db_transaction


@app.get("/transactions/", response_model=List[TransactionResponse])
def read_transactions():
    db = SessionLocal()
    transactions = db.query(Transaction).all()
    db.close()
    return transactions


@app.get("/transactions/{transaction_id}", response_model=TransactionResponse)
def read_transaction(transaction_id: int):
    db = SessionLocal()
    transaction = db.query(Transaction).filter(Transaction.id ==
transaction_id).first()
    db.close()
    if not transaction:
        raise HTTPException(status_code=404, detail="Transaction not found")
    return transaction


@app.post("/invoices/{invoice_id}/pay")
def pay_invoice(invoice_id: int):
    db = SessionLocal()

    invoice = db.query(Invoice).filter(Invoice.id == invoice_id).first()
    if not invoice:
        db.close()
```

```
        raise HTTPException(status_code=404, detail="Invoice not found")

    if invoice.status == InvoiceStatus.paid:
        db.close()
        raise HTTPException(status_code=400, detail="Invoice already paid")

    if invoice.status != InvoiceStatus.sent:
        db.close()
        raise HTTPException(status_code=400, detail="Invoice must be sent before
payment")

    revenue_account = db.query(Account).filter(
        Account.account_type == AccountType.revenue,
        Account.organization_id == invoice.organization_id
    ).first()

    if not revenue_account:
        db.close()
        raise HTTPException(status_code=400, detail="Revenue account not found")

    asset_account = db.query(Account).filter(
        Account.account_type == AccountType.asset,
        Account.organization_id == invoice.organization_id
    ).first()

    if not asset_account:
        db.close()
        raise HTTPException(status_code=400, detail="Asset account not found")

    revenue_transaction = Transaction(
        amount=invoice.total_amount,
```

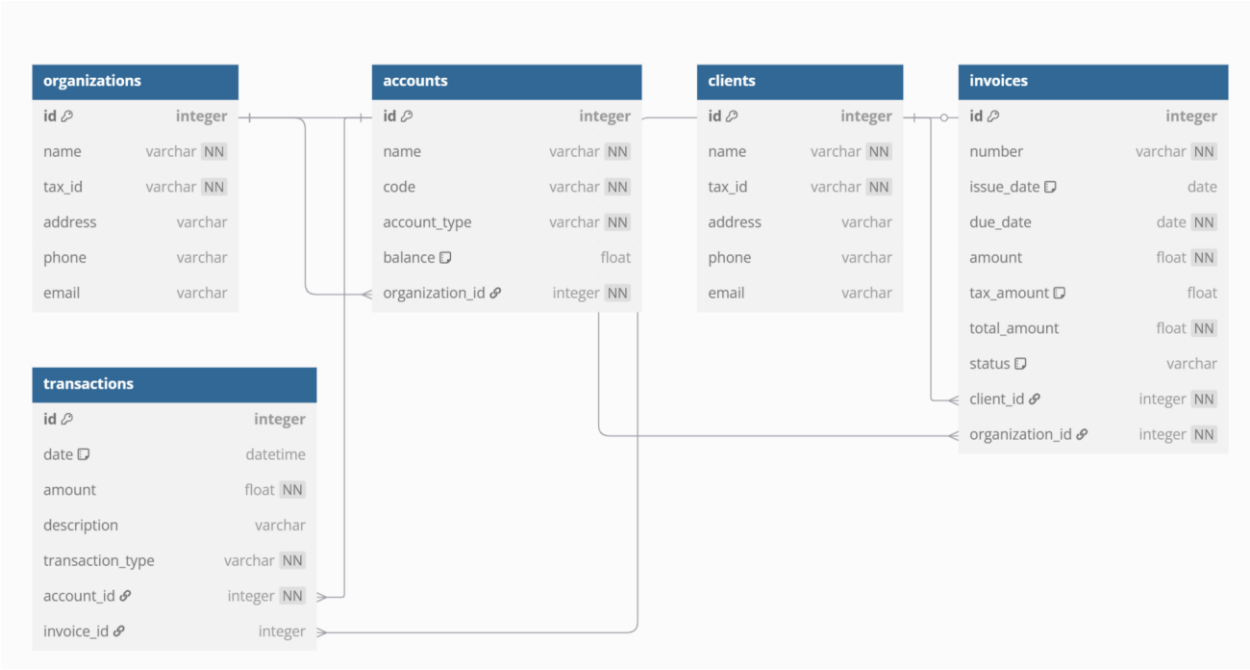
```
        transaction_type=TransactionType.credit,
        account_id=revenue_account.id,
        invoice_id=invoice.id,
        description=f"Payment for invoice {invoice.number}"
    )
    db.add(revenue_transaction)
    revenue_account.balance -= invoice.total_amount

    asset_transaction = Transaction(
        amount=invoice.total_amount,
        transaction_type=TransactionType.debit,
        account_id=asset_account.id,
        invoice_id=invoice.id,
        description=f"Payment received for invoice {invoice.number}"
    )
    db.add(asset_transaction)
    asset_account.balance += invoice.total_amount

    invoice.status = InvoiceStatus.paid

    db.commit()
    db.close()
    return {"message": "Invoice paid successfully"}
```

Схема:



Вывод: приобрел практические навыки разработки API и баз данных.