**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ   
РОССИЙСКОЙ ФЕДЕРАЦИИ**

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

**«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ**

**ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ им. В. Г. ШУХОВА»**

**(БГТУ им. В.Г. Шухова)**



ИНСТИТУТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И УПРАВЛЯЮЩИХ СИСТЕМ

**Лабораторная работа №7**

по дисциплине: Базы данных

тема: «Организация взаимодействия с базой данных через приложение, использующее технологию ORM»

Выполнил: ст. группы ПВ-223

Пахомов Владислав Андреевич

Проверили:

ст. пр. Панченко Максим Владимирович

Белгород 2024 г.

**Лабораторная работа №7**

Организация взаимодействия с базой данных через приложение, использующее технологию ORM  
Вариант 8

**Цель работы:** разработать приложение, использующее технологию ORM, для взаимодействия с базой данных.

Чтобы наше приложение было полноценным и было как можно ближе к пользователю, создадим файл для запуска нашего приложения, которое будет «компилировать» .ui файлы, загружать venv, устанавливать зависимости и выполнять миграцию для базы данных, считывать .env файл:

echo "> Read .env file"

# Default location of venv file

set-content env:\VENV\_DIR .venv

get-content .env | foreach {

    $name, $value = $\_.split('=')

    set-content env:\$name $value

}

if (!(Test-Path "$env:VENV\_DIR/Scripts/python.exe")){

   echo "> Venv is not found, let's install it. It'd take some time, grab some coffee!"

   if (Test-Path "$env:VENV\_DIR") {

        rm -r "$env:VENV\_DIR"

   }

   mkdir "$env:VENV\_DIR"

   python -m venv "$env:VENV\_DIR"

}

echo "> Installing requirments"

powershell "$env:VENV\_DIR/Scripts/pip.exe install -r ./requirments.txt"

echo "> Convert .ui files into .py files"

powershell "$env:VENV\_DIR/Lib/site-packages/PySide6/uic.exe widgets/main.ui -g python -o widgets/main\_ui.py"

powershell "$env:VENV\_DIR/Lib/site-packages/PySide6/uic.exe widgets/repotab.ui -g python -o widgets/repotab\_ui.py"

powershell "$env:VENV\_DIR/Scripts/alembic-autogen-check"

if (!($?)) {

    echo "> Database is not up-to-date"

    echo "> Autogen new revision"

    powershell "$env:VENV\_DIR/Scripts/alembic revision --autogenerate"

    echo "> Upgrade to latest alembic version"

    powershell "$env:VENV\_DIR/Scripts/alembic upgrade head"

} else {

    echo "> Database is up-to-date"

}

echo "> Start application"

powershell "$env:VENV\_DIR/Scripts/python.exe main.py"

В качестве ORM будем использовать SQLAlchemy в сочетании с Alembic для автогенерации миграций. Задачу по валидации данных перенесём из базы данных в схемы Pydantic. Он же и позволит нам «подтянуть» настройки окружения.

from pydantic\_settings import BaseSettings

class Settings(BaseSettings):

    POSTGRES\_DSN: str

"""

Настройки для приложения

"""

settings = Settings()

Здесь будем создавать движок для работы с базой данных

from sqlalchemy import create\_engine

from core.config import settings

from sqlalchemy.orm import declarative\_base

engine = create\_engine(settings.POSTGRES\_DSN)

Base = declarative\_base()

Alembic позволяет отслеживать изменения в моделях программы и автоматически генерировать миграции. Как было уже сказано ранее, перед каждым запуском приложения мы создаём новую миграцию, если нужно, и сразу же выполняем её, чтобы привести базу данных к актуальным моделям. Так, например, выглядит одна из миграций приложения:

"""empty message

Revision ID: bf4013dac0b6

Revises:

Create Date: 2024-12-05 15:06:01.721226

"""

from typing import Sequence, Union

from alembic import op

import sqlalchemy as sa

# revision identifiers, used by Alembic.

revision: str = 'bf4013dac0b6'

down\_revision: Union[str, None] = None

branch\_labels: Union[str, Sequence[str], None] = None

depends\_on: Union[str, Sequence[str], None] = None

def upgrade() -> None:

    # ### commands auto generated by Alembic - please adjust! ###

    op.create\_table('home',

    sa.Column('address', sa.String(), nullable=False),

    sa.Column('commisioning', sa.Date(), nullable=True),

    sa.Column('floors', sa.Integer(), nullable=False),

    sa.Column('index', sa.Integer(), nullable=False),

    sa.PrimaryKeyConstraint('address')

    )

    op.create\_table('resident',

    sa.Column('passport\_data', sa.String(), nullable=False),

    sa.Column('snp', sa.String(), nullable=False),

    sa.Column('email', sa.String(), nullable=False),

    sa.Column('phone', sa.String(), nullable=False),

    sa.PrimaryKeyConstraint('passport\_data')

    )

    op.create\_table('worker',

    sa.Column('inn', sa.String(), nullable=False),

    sa.Column('email', sa.String(), nullable=True),

    sa.Column('phone', sa.String(), nullable=True),

    sa.PrimaryKeyConstraint('inn')

    )

    op.create\_table('contract',

    sa.Column('id', sa.Integer(), nullable=False),

    sa.Column('transaction\_date', sa.Date(), nullable=False),

    sa.Column('until\_date', sa.Date(), nullable=False),

    sa.Column('payment', sa.Integer(), nullable=False),

    sa.Column('home\_address', sa.String(), nullable=False),

    sa.ForeignKeyConstraint(['home\_address'], ['home.address'], ondelete='restrict'),

    sa.PrimaryKeyConstraint('id')

    )

    op.create\_table('task',

    sa.Column('id', sa.Integer(), nullable=False),

    sa.Column('payment', sa.Integer(), nullable=False),

    sa.Column('completed\_date', sa.Date(), nullable=True),

    sa.Column('until\_date', sa.Date(), nullable=False),

    sa.Column('home\_address', sa.String(), nullable=False),

    sa.ForeignKeyConstraint(['home\_address'], ['home.address'], ondelete='restrict'),

    sa.PrimaryKeyConstraint('id')

    )

    op.create\_table('payment',

    sa.Column('id', sa.String(), nullable=False),

    sa.Column('paid\_date', sa.Date(), nullable=True),

    sa.Column('until\_date', sa.Date(), nullable=False),

    sa.Column('contract\_id', sa.Integer(), nullable=True),

    sa.ForeignKeyConstraint(['contract\_id'], ['contract.id'], ondelete='set null'),

    sa.PrimaryKeyConstraint('id')

    )

    op.create\_table('residents\_contracts',

    sa.Column('resident\_passport\_data', sa.String(), nullable=False),

    sa.Column('contract\_id', sa.Integer(), nullable=False),

    sa.ForeignKeyConstraint(['contract\_id'], ['contract.id'], ),

    sa.ForeignKeyConstraint(['resident\_passport\_data'], ['resident.passport\_data'], ),

    sa.PrimaryKeyConstraint('resident\_passport\_data', 'contract\_id')

    )

    op.create\_table('workers\_tasks',

    sa.Column('worker\_inn', sa.String(), nullable=False),

    sa.Column('task\_id', sa.Integer(), nullable=False),

    sa.ForeignKeyConstraint(['task\_id'], ['task.id'], ),

    sa.ForeignKeyConstraint(['worker\_inn'], ['worker.inn'], ),

    sa.PrimaryKeyConstraint('worker\_inn', 'task\_id')

    )

    # ### end Alembic commands ###

def downgrade() -> None:

    # ### commands auto generated by Alembic - please adjust! ###

    op.drop\_table('workers\_tasks')

    op.drop\_table('residents\_contracts')

    op.drop\_table('payment')

    op.drop\_table('task')

    op.drop\_table('contract')

    op.drop\_table('worker')

    op.drop\_table('resident')

    op.drop\_table('home')

    # ### end Alembic commands ###

Классы DTO генераторы задают, какие из методов CRUD будут доступны, какие данные им будут нужны и какие данные они будут задавать, а также позволяет локализировать поля, позволяя поверхностно настраивать репозиторий:

from dto.base import BaseDTOGeneartor

from schemas.payment import PaymentCreate, PaymentUpdate, PaymentIdentifier, PaymentShow

class PaymentDTOGenerator(BaseDTOGeneartor):

    def translations(self):

        return {

            "id": "УИП",

            "paid\_date": "Дата оплаты",

            "until\_date": "Срок оплаты",

            "contract\_id": "Ид. ном. договора",

            "energy\_source": "Энергетический ресурс",

            "payment": "Сумма"

        }

    def select(self):

        return PaymentShow

    def insert(self):

        return PaymentCreate

    def update(self):

        return PaymentUpdate

    def identifier(self):

        return PaymentIdentifier

Репозиторий, или менеджер, уже на основе данных генератора выполняет сами запросы:

from sqlalchemy import select, ScalarResult, and\_

from tabulate import tabulate

from typing import Any

from core.db import engine

from dto.base import BaseDTOGeneartor

from sqlalchemy.orm import Session

from schemas.base import TunedModel

class Repository:

    def \_\_init\_\_(self, table, generator: BaseDTOGeneartor):

        self.\_table = table

        self.generator = generator

    def get\_dto\_generator(self):

        return self.generator

    def select(self) -> list[TunedModel]:

        select\_model = self.generator.select()

        if select\_model is None:

            raise NotImplementedError(f"Выбрать из {self.\_table.\_\_tablename\_\_} невозможно")

        with Session(engine) as session:

            results = session.execute(select(self.\_table)).all()

            return [select\_model.model\_validate(result[0]) for result in results]

    def insert(self, data: dict) -> None:

        insert\_model = self.generator.insert()

        if insert\_model is None:

            raise NotImplementedError(f"Вставить в {self.\_table.\_\_tablename\_\_} невозможно")

        insert\_data = insert\_model(\*\*data)

        with Session(engine) as session:

            new\_object = self.\_table(\*\*insert\_data.dict())

            session.add(new\_object)

            session.commit()

    def update(self, data: dict, identifier: dict) -> None:

        update\_model = self.generator.update()

        identifier\_model = self.generator.identifier()

        if update\_model is None or identifier\_model is None:

            raise NotImplementedError(f"Обновить {self.\_table.\_\_tablename\_\_} невозможно")

        update\_data = update\_model(\*\*data)

        identifier\_data = identifier\_model(\*\*identifier)

        with Session(engine) as session:

            object = session.execute(select(self.\_table).filter(

                and\_(

                    \*(list(map(lambda x: getattr(self.\_table, x[0]) == x[1], identifier\_data.model\_dump().items())))

                )

            )).one\_or\_none()

            if not object:

                raise LookupError(f"Элемент {self.\_table.\_\_tablename\_\_} с {identifier\_data.model\_dump()} невозможно найти")

            object = object[0]

            for key, value in update\_data.model\_dump().items():

                setattr(object, key, value)

            session.commit()

    def delete(self, identifier: dict) -> None:

        identifier\_model = self.generator.identifier()

        if identifier\_model is None:

            raise NotImplementedError(f"Удалить {self.\_table.\_\_tablename\_\_} невозможно")

        identifier\_data = identifier\_model(\*\*identifier)

        with Session(engine) as session:

            object = session.execute(select(self.\_table).filter(

                and\_(

                    \*(list(map(lambda x: getattr(self.\_table, x[0]) == x[1], identifier\_data.model\_dump().items())))

                )

            )).one\_or\_none()

            if not object:

                raise LookupError(

                    f"Элемент {self.\_table.\_\_tablename\_\_} с {identifier\_data.model\_dump()} невозможно найти")

            object = object[0]

            session.delete(object)

            session.commit()

Однако же если от репозитория нужно более сложное поведение, его можно переопределить:

from sqlalchemy import select, func, and\_, desc

from sqlalchemy.orm import Session

from core.db import engine

from dto import WorkrsRatingDTOGenerator

from models import Worker, WorkerTask, Task

from repositories.base import Repository

class WorkersRatingRepository(Repository):

    def \_\_init\_\_(self):

        super().\_\_init\_\_(None, WorkrsRatingDTOGenerator())

    def select(self):

        select\_model = self.generator.select()

        if select\_model is None:

            raise NotImplementedError(f"Выбрать из {self.\_table.\_\_tablename\_\_} невозможно")

        with Session(engine) as session:

            begin\_date = '2004-01-01'

            end\_date = '2040-01-01'

            completed = (

                select(

                    WorkerTask.worker\_inn,

                    func.count().label("completed")

                )

                .select\_from(WorkerTask)

                .join(Task)

                .where(

                    and\_(

                        Task.completed\_date.isnot(None),

                        Task.until\_date > begin\_date,

                        Task.until\_date < end\_date

                    )

                )

                .group\_by(WorkerTask.worker\_inn)

            ).subquery()

            total = (

                select(

                    WorkerTask.worker\_inn,

                    func.count().label("total")

                )

                .select\_from(WorkerTask)

                .join(Task)

                .where(

                    and\_(

                        Task.until\_date > begin\_date,

                        Task.until\_date < end\_date

                    )

                )

                .group\_by(WorkerTask.worker\_inn)

            ).subquery()

            results = session.execute(

                select(

                    Worker.inn.label("worker\_inn"),

                    func.coalesce(completed.c.completed, 0).label("completed"),

                    (1.0 \* func.coalesce(completed.c.completed, 0) / total.c.total).label("rating")

                )

                .select\_from(Worker)

                .join(completed, completed.c.worker\_inn == Worker.inn, isouter=True)

                .join(total, total.c.worker\_inn == Worker.inn)

                .order\_by(desc("completed"))

            )

            return [select\_model.model\_validate({"inn": result[0], "completed": result[1], "rating": result[2]}) for result in results]

    def insert(self, data: dict) -> None:

        return super().insert(data)

    def update(self, data: dict, identifier: dict) -> None:

        return super().update(data, identifier)

    def delete(self, identifier: dict) -> None:

        return super().delete(identifier)

from sqlalchemy import select

from sqlalchemy import func

from sqlalchemy.orm import Session

from core.db import engine

from dto import NonPayersDTOGenerator

from models import Resident, Payment, Contract, ResidentContract

from repositories.base import Repository

class NonPayersRepository(Repository):

    def \_\_init\_\_(self):

        super().\_\_init\_\_(None, NonPayersDTOGenerator())

    def select(self):

        select\_model = self.generator.select()

        if select\_model is None:

            raise NotImplementedError(f"Выбрать из {self.\_table.\_\_tablename\_\_} невозможно")

        with Session(engine) as session:

            results = session.execute(

                select(

                    func.concat(Resident.surname, " ", Resident.name, " ", Resident.patronymics).label("snp"),

                    func.sum(Payment.payment).label("debt"),

                    Payment.energy\_source

                )

                .select\_from(Resident)

                .join(ResidentContract, ResidentContract.resident\_passport\_data == Resident.passport\_data)

                .join(Contract)

                .join(Payment)

                .where(Payment.paid\_date.is\_(None))

                .group\_by(Resident.passport\_data, Payment.energy\_source)

                .order\_by("debt")

            ).all()

            return [select\_model.model\_validate({"snp": result[0], "debt": result[1], "energy\_source": result[2]}) for result in results]

    def insert(self, data: dict) -> None:

        return super().insert(data)

    def update(self, data: dict, identifier: dict) -> None:

        return super().update(data, identifier)

    def delete(self, identifier: dict) -> None:

        return super().delete(identifier)

from sqlalchemy import select, func

from sqlalchemy.orm import Session

from core.db import engine

from dto import HomeProfitDTOGenerator

from models import Home, Contract, Payment, Task

from repositories.base import Repository

class ProfitHouseRepository(Repository):

    def \_\_init\_\_(self):

        super().\_\_init\_\_(None, HomeProfitDTOGenerator())

    def select(self):

        select\_model = self.generator.select()

        if select\_model is None:

            raise NotImplementedError(f"Выбрать из {self.\_table.\_\_tablename\_\_} невозможно")

        with Session(engine) as session:

            plus = (select(

                Contract.home\_address.label("home"),

                func.sum(Payment.payment).label("plus")

            )

                    .select\_from(Payment)

                    .join(Contract, isouter=True)

                    .group\_by(Contract.home\_address)

                    .subquery())

            minus = (select(

                Task.home\_address.label("home"),

                func.sum(Task.payment).label("minus")

            )

                     .select\_from(Task)

                     .group\_by(Task.home\_address)

                     .subquery())

            results = session.execute(

                (select(

                    Home.address.label("address"),

                    (

                            func.coalesce(plus.c.plus, 0) - func.coalesce(minus.c.minus, 0)

                    ).label("profit")

                ).select\_from(Home)

                 .join(plus, plus.c.home == Home.address, isouter=True)

                 .join(minus, minus.c.home == Home.address, isouter=True)

                 .order\_by("profit"))

            ).all()

            return [select\_model.model\_validate({"address": result[0], "profit": result[1]}) for result in results]

    def insert(self, data: dict) -> None:

        return super().insert(data)

    def update(self, data: dict, identifier: dict) -> None:

        return super().update(data, identifier)

    def delete(self, identifier: dict) -> None:

        return super().delete(identifier)

Схемы, или же сериализаторы, выполняют роль валидации данных, а также задают структуру ответов и запросов

from typing import Optional

from pydantic import field\_validator

from schemas.base import TunedModel, RuNumberType

from pydantic import EmailStr

import re

class ResidentIdentifier(TunedModel):

    passport\_data: str

    @field\_validator("passport\_data", mode="after")

    def validate\_passport\_data(cls, value):

        if not re.match(r"^\d{10}$", value):

            raise ValueError("Паспортные данные должны содержать 10 цифр, 4 первые - серия, 6 остальных - номер паспорта")

        return value

class ResidentUpdate(TunedModel):

    surname: str

    name: str

    patronymics: Optional[str]

    email: Optional[EmailStr]

    phone: Optional[RuNumberType]

    @field\_validator("surname", "name", mode="after")

    def validate\_surname\_name(cls, value):

        if not re.match(r"^[А-ЯЁа-яёA-Za-z-]+$", value):

            raise ValueError("Фамилия и имя должны содержать латинские, кириллические символы и дефис, они не должны быть пустыми")

        return value

    @field\_validator("surname", "name", mode="after")

    def validate\_patronymics(cls, value):

        if value is None:

            return value

        if not re.match(r"^[А-ЯЁа-яёA-Za-z-]+$", value):

            raise ValueError("Отчество должно содержать латинские, кириллические символы и дефис")

        return value

class ResidentCreate(ResidentUpdate, ResidentIdentifier):

    pass

class ResidentShow(ResidentCreate):

    pass

Repotab, или же сервис, взаимодействует с репозиторием (или же менеджером) и выполняет отображение для клиента

import PySide6.QtCore

import PySide6.QtWidgets

from PySide6.QtWidgets import QWidget, QTableWidgetItem, QDialog

from repositories.base import Repository

from widgets.accept\_reject import AcceptRejectDialog

from widgets.form\_dialog import FormDialog

from widgets.repotab\_ui import Ui\_Form

class RepoTab(QWidget):

    def \_\_init\_\_(self, repository: Repository, parent=None) -> None:

        super().\_\_init\_\_(parent)

        self.ui = Ui\_Form()

        self.ui.setupUi(self)

        self.\_\_repository = repository

        self.\_\_values = []

        # Прячем вставку, если DTO вставки пустой

        if self.\_\_repository.generator.insert() is None:

            self.ui.pushButton.setVisible(False)

        # Прячем обновление, если нечего обновляьт или нельзя идентифицировать запись

        if self.\_\_repository.generator.identifier() is None or self.\_\_repository.generator.update() is None:

            self.ui.pushButton\_2.setVisible(False)

        # Прячем удаление, если нельзя идентифицировать запись

        if self.\_\_repository.generator.identifier() is None:

            self.ui.pushButton\_3.setVisible(False)

        self.ui.pushButton\_3.clicked.connect(self.\_\_delete\_clicked)

        self.ui.pushButton\_2.clicked.connect(self.\_\_update\_clicked)

        self.ui.pushButton.clicked.connect(self.\_\_create\_clicked)

        self.refetch\_table()

    def refetch\_table(self):

        self.\_\_values = self.\_\_repository.select()

        self.\_\_redraw\_table()

    def \_\_redraw\_table(self):

        translations = self.\_\_repository.generator.translations()

        select\_keys = list(self.\_\_repository.generator.select().model\_fields.keys())

        self.ui.tableWidget.setColumnCount(len(select\_keys))

        self.ui.tableWidget.setRowCount(len(self.\_\_values))

        self.ui.tableWidget.setHorizontalHeaderLabels(list(map(lambda x: translations[x], select\_keys)))

        i = 0

        for value in self.\_\_values:

            j = 0

            show\_model = value.model\_dump()

            for key in select\_keys:

                self.ui.tableWidget.setItem(i, j, QTableWidgetItem("Пусто" if show\_model[key] is None else str(show\_model[key])))

                j += 1

            i += 1

        self.ui.tableWidget.resizeColumnsToContents()

    def \_\_create\_clicked\_callback(self, parent\_dialog, data):

        try:

            self.\_\_repository.insert(data)

            self.refetch\_table()

            return True

        except Exception as e:

            whoops = AcceptRejectDialog(parent=parent\_dialog,

                                        title="Произошла ошибка",

                                        text=repr(e))

            whoops.show()

            return False

    def \_\_create\_clicked(self):

        form\_dialog = FormDialog("Создать",

                                 list(self.\_\_repository.generator.insert().model\_fields.keys()),

                                 self.\_\_repository.generator.translations(),

                                 self,

                                 self.\_\_create\_clicked\_callback)

        form\_dialog.exec()

    def \_\_delete\_clicked(self):

        should\_delete = AcceptRejectDialog(parent=self,

                                           title="Удалить?",

                                           text=f"Вы собираетесь удалить записи ({len(self.ui.tableWidget.selectionModel().selectedRows())})")

        if should\_delete.exec() == 1:

            for row\_index in self.ui.tableWidget.selectionModel().selectedRows():

                select\_keys = self.\_\_repository.generator.select().model\_fields.keys()

                selected\_value: dict = {}

                for key, val in self.\_\_values[row\_index.row()].\_\_dict\_\_.items():

                    if key in select\_keys:

                        selected\_value[key] = val

                selected\_value\_identififer\_keys = list(self.\_\_repository.generator.identifier().model\_fields.keys())

                selected\_value\_identififer: dict = {}

                select\_keys = list(self.\_\_repository.generator.select().model\_fields.keys())

                for key\_index in range(0, len(select\_keys)):

                    if select\_keys[key\_index] in selected\_value\_identififer\_keys:

                        selected\_value\_identififer[select\_keys[key\_index]] = selected\_value[select\_keys[key\_index]]

                try:

                    self.\_\_repository.delete(selected\_value\_identififer)

                except Exception as e:

                    whoops = AcceptRejectDialog(parent=self,

                                                title="Произошла ошибка",

                                                text=repr(e))

                    whoops.show()

            self.refetch\_table()

    def \_\_update\_clicked\_callback(self, parent\_dialog, data):

        row\_index = self.ui.tableWidget.selectionModel().selectedRows()[0]

        select\_keys = self.\_\_repository.generator.select().model\_fields.keys()

        selected\_value: dict = {}

        for key, val in self.\_\_values[row\_index.row()].\_\_dict\_\_.items():

            if key in select\_keys:

                selected\_value[key] = val

        selected\_value\_identififer\_keys = list(self.\_\_repository.generator.identifier().model\_fields.keys())

        selected\_value\_identififer: dict = {}

        select\_keys = list(self.\_\_repository.generator.select().model\_fields.keys())

        for key\_index in range(0, len(select\_keys)):

            if select\_keys[key\_index] in selected\_value\_identififer\_keys:

                selected\_value\_identififer[select\_keys[key\_index]] = selected\_value[select\_keys[key\_index]]

        try:

            self.\_\_repository.update(data, selected\_value\_identififer)

            self.refetch\_table()

            return True

        except Exception as e:

            whoops = AcceptRejectDialog(parent=parent\_dialog,

                                        title="Произошла ошибка",

                                        text=repr(e))

            whoops.show()

            return False

    def \_\_update\_clicked(self):

        if len(self.ui.tableWidget.selectionModel().selectedRows()) == 0:

            no\_update = AcceptRejectDialog(parent=self,

                                           title="Нечего обновлять",

                                           text=f"Выберите один ряд для обновления")

            no\_update.exec()

            return

        row\_index = self.ui.tableWidget.selectionModel().selectedRows()[0]

        select\_keys = self.\_\_repository.generator.select().model\_fields.keys()

        selected\_value: dict = {}

        for key, val in self.\_\_values[row\_index.row()].\_\_dict\_\_.items():

            if key in select\_keys:

                selected\_value[key] = val

        selected\_value\_identififer\_keys = list(self.\_\_repository.generator.identifier().model\_fields.keys())

        selected\_value\_identififer: dict = {}

        select\_keys = list(self.\_\_repository.generator.select().model\_fields.keys())

        for key\_index in range(0, len(select\_keys)):

            if select\_keys[key\_index] in selected\_value\_identififer\_keys:

                selected\_value\_identififer[select\_keys[key\_index]] = selected\_value[select\_keys[key\_index]]

        form\_dialog = FormDialog("Обновить",

                                 list(self.\_\_repository.generator.update().model\_fields.keys()),

                                 self.\_\_repository.generator.translations(),

                                 self,

                                 self.\_\_update\_clicked\_callback,

                                 selected\_value)

        form\_dialog.exec()

main.py создаёт вкладки на основе репозиториев

from PySide6.QtWidgets import QMainWindow

from repositories.repositories import all\_repos

from widgets.main\_ui import Ui\_MainWindow

from widgets.repotab import RepoTab

class MainDialog(QMainWindow):

    def \_\_init\_\_(self, parent=None):

        super().\_\_init\_\_(parent=parent)

        self.ui = Ui\_MainWindow()

        self.ui.setupUi(self)

        self.\_\_tabs = []

        for repo\_item in all\_repos:

            tab\_test = RepoTab(repo\_item["repo"])

            self.\_\_tabs.append(tab\_test)

            self.ui.tabs.addTab(tab\_test, repo\_item["name"])

        self.ui.tabs.tabBarClicked.connect(self.\_\_upadte\_tab\_on\_select)

    def \_\_upadte\_tab\_on\_select(self, index):

        self.\_\_tabs[index].refetch\_table()

from dto import ContractDTOGenerator, HomeDTOGenerator, PaymentDTOGenerator, ResidentDTOGenerator, \

    ResidentContractDTOGenerator, TaskDTOGenerator, WorkerDTOGenerator, WorkerTaskDTOGenerator

from models import Contract, Home, Payment, Resident, Task, Worker, WorkerTask, ResidentContract

from repositories.base import Repository

from repositories.non\_payers\_repository import NonPayersRepository

from repositories.profit\_house\_repository import ProfitHouseRepository

from repositories.workers\_rating\_repository import WorkersRatingRepository

contract\_repository = Repository(Contract, ContractDTOGenerator())

home\_repository = Repository(Home, HomeDTOGenerator())

payment\_repository = Repository(Payment, PaymentDTOGenerator())

resident\_repository = Repository(Resident, ResidentDTOGenerator())

residents\_contracts\_repository = Repository(ResidentContract, ResidentContractDTOGenerator())

task\_repository = Repository(Task, TaskDTOGenerator())

worker\_repository = Repository(Worker, WorkerDTOGenerator())

workers\_tasks\_repository = Repository(WorkerTask, WorkerTaskDTOGenerator())

non\_payers\_repository = NonPayersRepository()

workers\_rating\_repository = WorkersRatingRepository()

profit\_house\_repository = ProfitHouseRepository()

all\_repos = [

    {"repo": contract\_repository, "name": "Договоры"},

    {"repo": home\_repository, "name": "Дома"},

    {"repo": payment\_repository, "name": "Чеки"},

    {"repo": resident\_repository, "name": "Жильцы"},

    {"repo": residents\_contracts\_repository, "name": "Договоры жильцов"},

    {"repo": task\_repository, "name": "Работы"},

    {"repo": worker\_repository, "name": "Исполнители работ"},

    {"repo": workers\_tasks\_repository, "name": "Назначения работ"},

    {"repo": non\_payers\_repository, "name": "Жильцы-неплательщики"},

    {"repo": workers\_rating\_repository, "name": "Рейтинг рабочих"},

    {"repo": profit\_house\_repository, "name": "Прибыль домов"}

]

В пакете models можно описать модель

from core.db import Base

from typing import TYPE\_CHECKING, List

from sqlalchemy.orm import Mapped, mapped\_column, relationship

from datetime import date

from sqlalchemy.sql.schema import ForeignKey

if TYPE\_CHECKING:

    from models import (  # noqa: F401

        Home,

        Worker

    )

class Task(Base):

    \_\_tablename\_\_ = "task"

    id: Mapped[int] = mapped\_column(primary\_key=True)

    payment: Mapped[int]

    completed\_date: Mapped[date] = mapped\_column(nullable=True)

    until\_date: Mapped[date]

    home\_address: Mapped[str] = mapped\_column(

        ForeignKey("home.address", ondelete="restrict"), nullable=False

    )

    home: Mapped["Home"] = relationship(back\_populates="tasks")

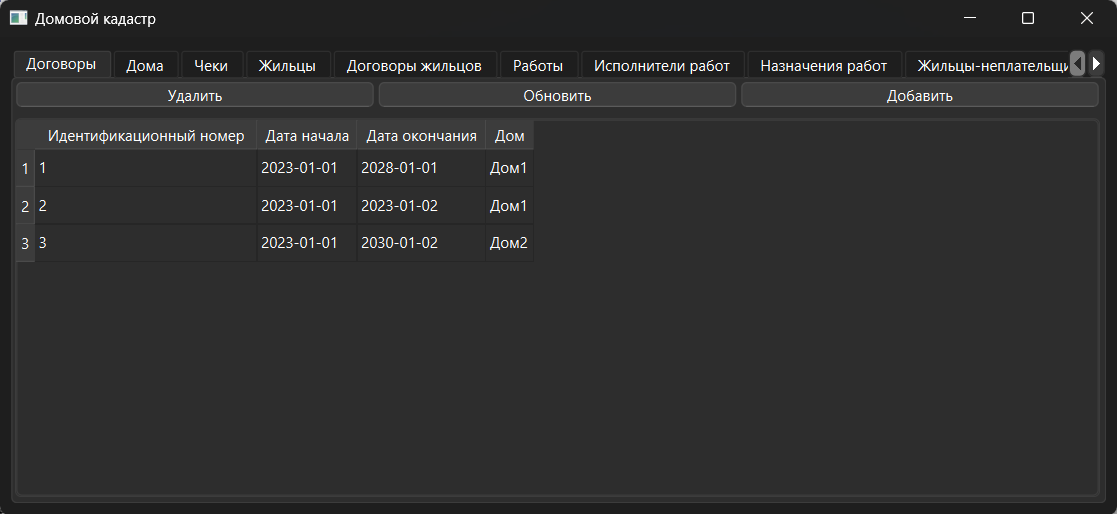
    workers: Mapped[List["Worker"]] = relationship(

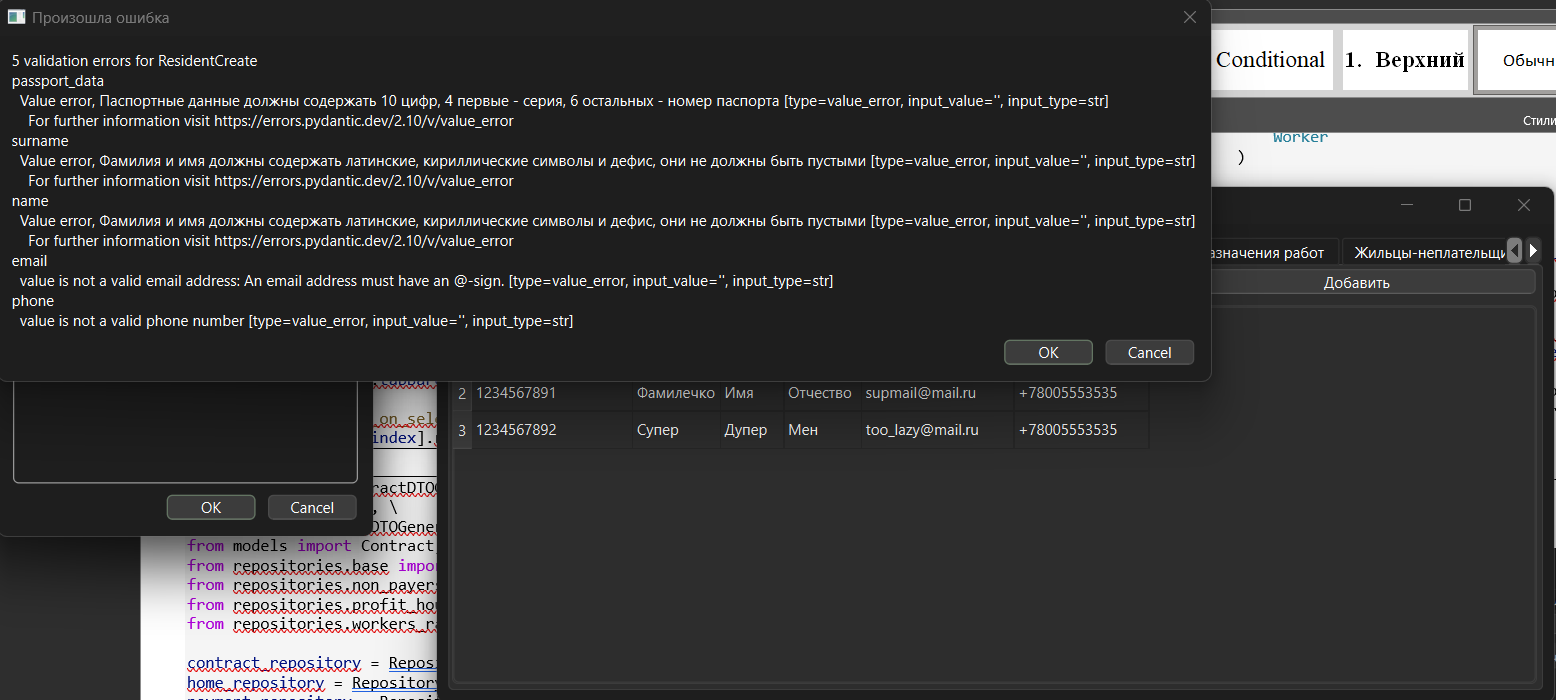
        secondary="workers\_tasks",

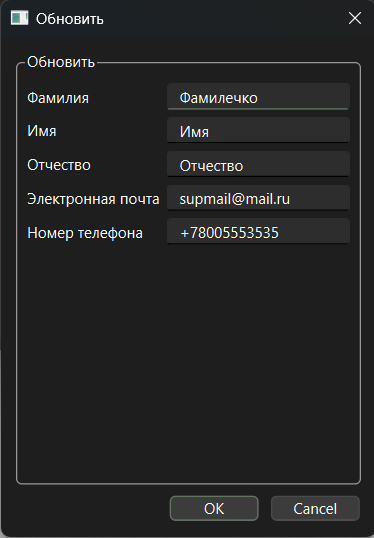
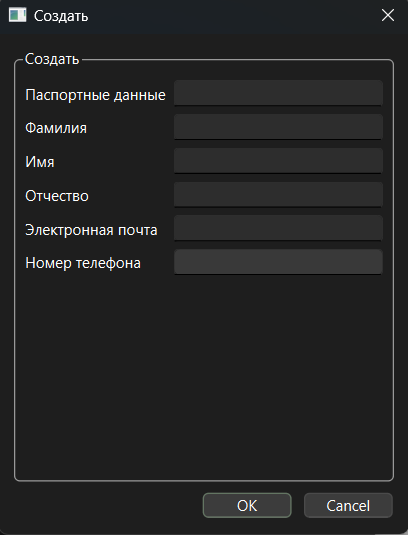
        back\_populates="tasks"

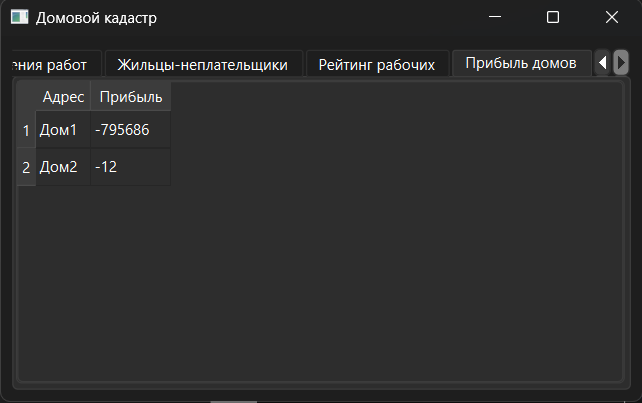
    )

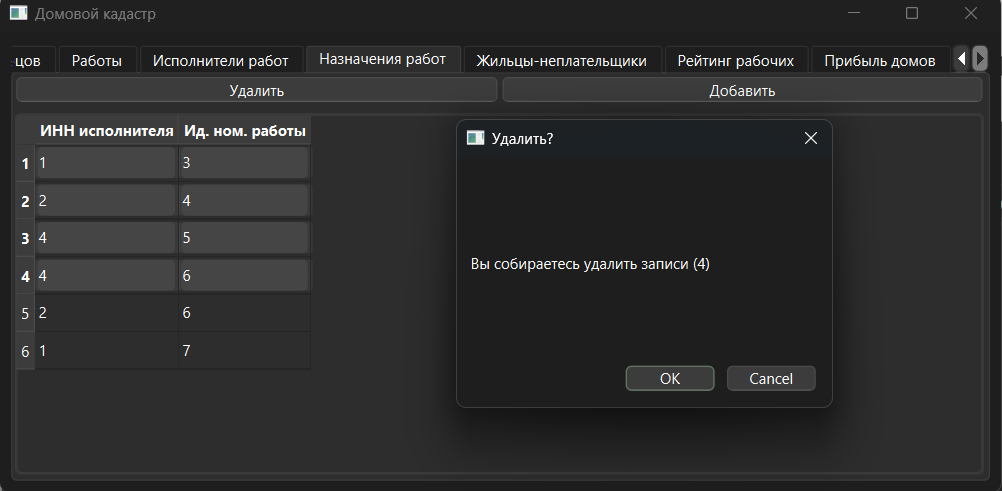
Скриншоты:











Ссылка на приложение: <https://github.com/IAmProgrammist/database/tree/main/lab7>

**Вывод:** в ходе лабораторной работы разработали приложение, использующее технологию ORM, для взаимодействия с базой данных.