

**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('commissioning', 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 локализовать 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"^[A-ЯЁа-яЁА-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"^[A-ЯЁа-яЁА-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 обновлять 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_identifer_keys =
list(self.__repository.generator.identifier().model_fields.keys())
            selected_value_identifer: 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_identifer_keys:
                    selected_value_identifer[select_keys[key_index]] =
selected_value[select_keys[key_index]]
            try:
                self.__repository.delete(selected_value_identifer)
            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_identifer_keys =
list(self.__repository.generator.identifier().model_fields.keys())
    selected_value_identifer: 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_identifer_keys:
            selected_value_identifer[select_keys[key_index]] =
selected_value[select_keys[key_index]]

    try:
        self.__repository.update(data, selected_value_identifer)
        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_identifer_keys =
list(self.__repository.generator.identifier().model_fields.keys())
    selected_value_identifer: 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_identifer_keys:
            selected_value_identifer[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.__update_tab_on_select)  
  
    def __update_tab_on_select(self, index):  
        self.__tabs[index].refresh_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

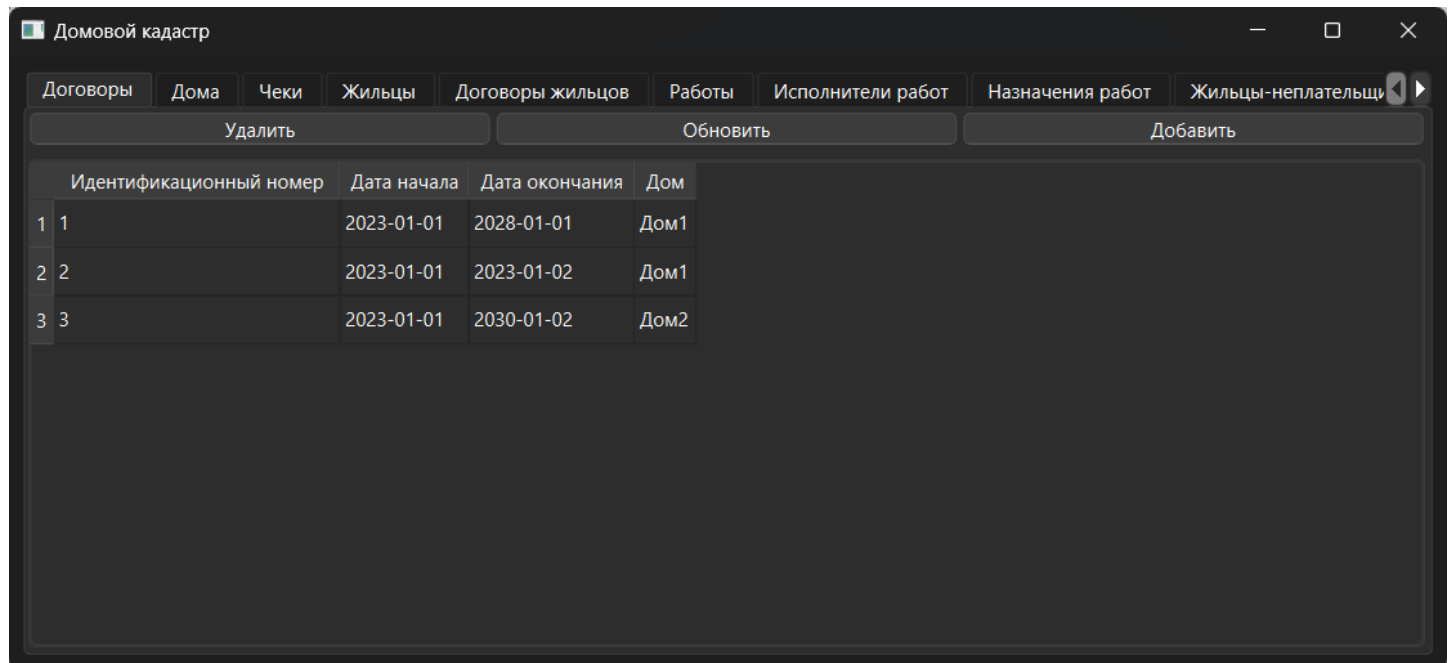
from typing import TYPE_CHECKING, List
from sqlalchemy.orm import Mapped, mapped_column, relationship
from datetime import date
from sqlalchemy.sql.schema import ForeignKey

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:



	Идентификационный номер	Дата начала	Дата окончания	Дом
1	1	2023-01-01	2028-01-01	Дом1
2	2	2023-01-01	2023-01-02	Дом1
3	3	2023-01-01	2030-01-02	Дом2

Произошла ошибка

5 validation errors for ResidentCreate  
passport\_data  
Value error, Паспортные данные должны содержать 10 цифр, 4 первые - серия, 6 остальных - номер паспорта [type=value\_error, input\_value="", input\_type=str]  
For further information visit https://errors.pydantic.dev/2.10/v/value\_error  
surname  
Value error, Фамилия и имя должны содержать латинские, кириллические символы и дефис, они не должны быть пустыми [type=value\_error, input\_value="", input\_type=str]  
For further information visit https://errors.pydantic.dev/2.10/v/value\_error  
name  
Value error, Фамилия и имя должны содержать латинские, кириллические символы и дефис, они не должны быть пустыми [type=value\_error, input\_value="", input\_type=str]  
For further information visit https://errors.pydantic.dev/2.10/v/value\_error  
email  
value is not a valid email address: An email address must have an @-sign. [type=value\_error, input\_value="", input\_type=str]  
phone  
value is not a valid phone number [type=value\_error, input\_value="", input\_type=str]

OKCancel

	2	1234567891	Фамиличко	Имя	Отчество	supmail@mail.ru	+78005553535
	3	1234567892	Супер	Дупер	Мен	too_lazy@mail.ru	+78005553535

OK

Cancel

```
from models import Contract
from repositories.base import BaseRepository
from repositories.non_payer import NonPayerRepository
from repositories.profit_holders import ProfitHoldersRepository
from repositories.workers import WorkersRepository

contract_repository = Repository[Contract]
home_repository = Repository[Home]
```

Conditional1. ВерхнийОбычный

worker

значения работЖильцы-неплательщи

Добавить

Создать

Создать

Паспортные данные

Фамилия

Имя

Отчество

Электронная почта

Номер телефона

OK

Cancel

Обновить

Обновить

Фамилия

Имя

Отчество

Электронная почта

Номер телефона

Фамиличко

Имя

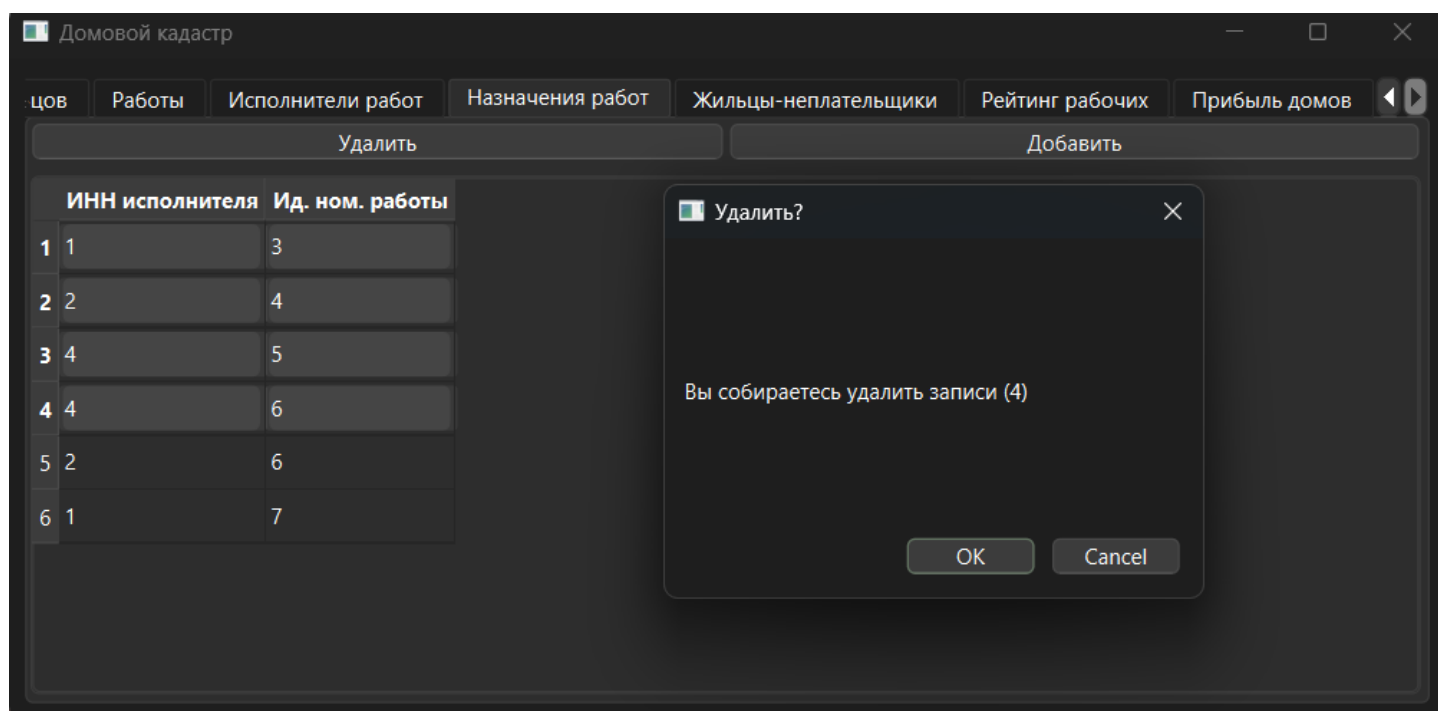
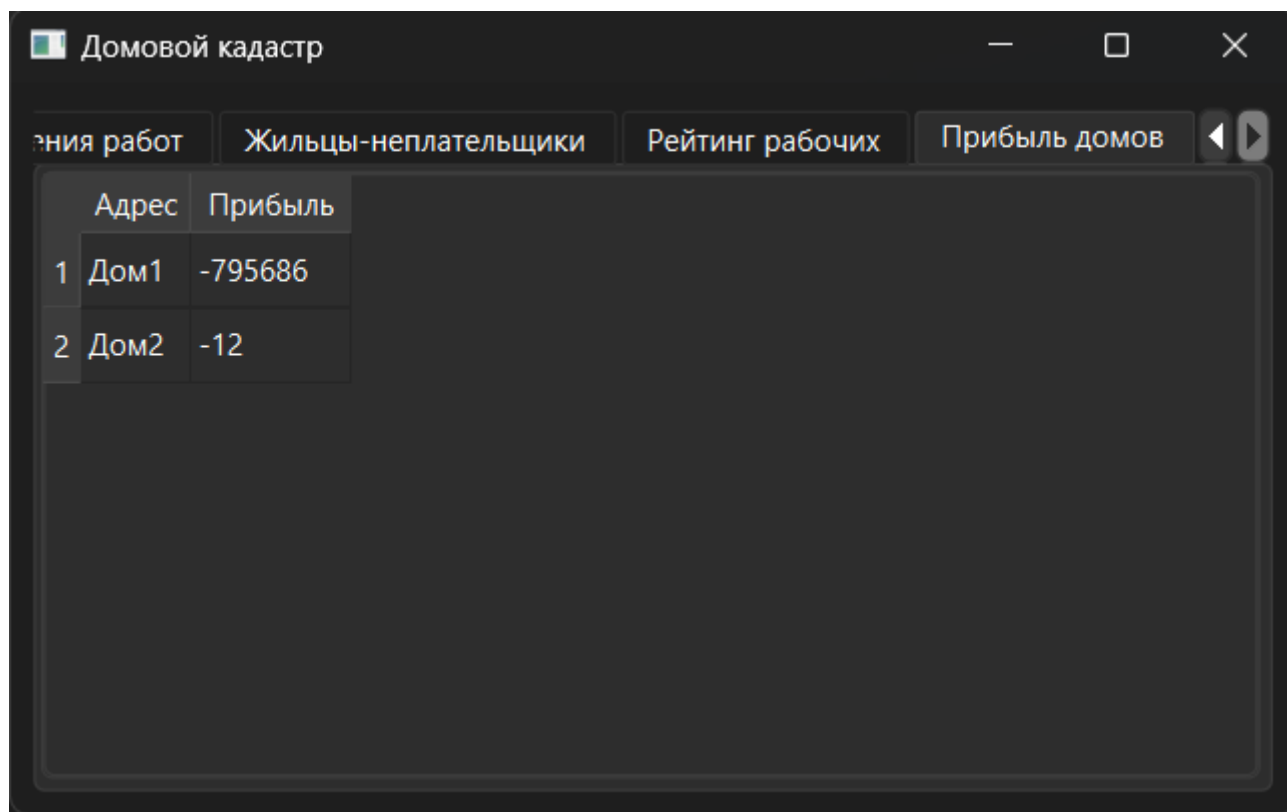
Отчество

supmail@mail.ru

+78005553535

OK

Cancel



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.