

**Министерство науки и высшего образования Российской Федерации  
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ  
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО**

**Факультет безопасности информационных технологий**

**Направление подготовки: 11.03.03**

**Образовательная программа: Безопасность информационных технологий**

**Дисциплина:**

«Информационная безопасность баз данных»

**ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №4**

«Реализация сервиса для взаимодействия с разработанной базой данных»

**Выполнил:**

Рядовой Т.С., студент группы N3352, поток ИББД.N63 1.5

\_\_\_\_\_ (подпись)

**Проверил:**

Салихов Максим Русланович

\_\_\_\_\_ (отметка о выполнении)

\_\_\_\_\_ (подпись)

\_\_\_\_\_ (дата)

Санкт-Петербург  
2025 г.

## СОДЕРЖАНИЕ

Введение.....	3
1    Ход работы.....	4
1.1    Обоснование выбора среды программирования и фреймворка .....	4
1.2    Структура.....	4
1.2.1    Микросервис db .....	5
1.2.2    Микросервис api .....	6
1.2.3    Микросервис nginx .....	7
1.2.4    docker-compose.....	7
1.3    Взаимодействие с веб-приложением и базой данных.....	8
1.4    Методы защиты данных.....	12
Заключение.....	13
Список источников .....	14
ПРИЛОЖЕНИЕ А.....	15

## **ВВЕДЕНИЕ**

Цель работы: получение навыков использования серверных языков программирования, фреймворков для работы с БД, ORM-систем.

# 1 ХОД РАБОТЫ

Задание:

1. На основе БД, созданной в предыдущих лабораторных работах, разработать сервис, который взаимодействует с разработанной БД.
2. Описать, каким образом происходит взаимодействие с базой данных, с помощью каких фреймворков или языков. Продемонстрировать примеры выборки, вставки, удаления данных из БД.
3. Описать основные функции сервиса, его назначение и структуру. Оценить методы защиты данных, реализованные в сервисе.

## 1.1 Обоснование выбора среды программирования и фреймворка

Для реализации сервиса взаимодействия с базой данных выбран язык программирования Python и фреймворк FastAPI. Все сервисы упакованы в Docker контейнеры для упрощения и безопасности разработки. Запуск производится посредством команды docker-compose. Фронтенд написан с использованием html и JavaScript.

Обоснование выбора:

- Python — простой и мощный язык, широко используемый для разработки веб-приложений и работы с базами данных;
- FastAPI — набирающий все большую популярность фреймворк для создания веб-приложений, который позволяет быстро разрабатывать API и веб-интерфейсы;
- Для работы с базой данных используется ORM SQLAlchemy, который упрощает взаимодействие с БД и позволяет избежать написания сложных SQL-запросов.

## 1.2 Структура

Всего 3 микросервиса:

- db: база данных PostgreSQL;
- api: основная логика бэкенда и фронтенда;
- nginx: веб-сервер для проксирования запросов.

### 1.2.1 Микросервис db

Вся инициализация базы данных из предыдущих лабораторных работ разбита на файлы с расширением sql. Также добавлена таблица users, где хранятся пользователи и их захешированные пароли. Для достижения этого использовалась хэш-функция bcrypt.

#### Листинг 1 – Таблица пользователей

```
class User(Base):
    __tablename__ = "users"
    id = Column(Integer, primary_key=True, index=True)
    email = Column(String, unique=True, index=True)
    hashed_password = Column(String)
```

#### Листинг 2 – Создание хэша пароля в файле utils.py

```
from passlib.context import CryptContext #type: ignore

# Hashing algorithm for passwords
pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")

# Function to hash passwords
def get_password_hash(password: str) -> str:
    return pwd_context.hash(password)

# Function to verify a hashed password
def verify_password(plain_password: str, hashed_password: str) -> bool:
    return pwd_context.verify(plain_password, hashed_password)
```

```
> psql -h localhost -p 5433 -U postgres -d repair_workshop
Пароль пользователя postgres:
psql (16.4 (Homebrew), сервер 15.12 (Debian 15.12-1.pgdg120+1))
Введите "help", чтобы получить справку.

repair_workshop=# \dt
          Список отношений
 Схема |      Имя      | Тип   | Владелец
-----+-----+-----+-----
public | clients       | таблица | postgres
public | employees     | таблица | postgres
public | main_log      | таблица | postgres
public | materials     | таблица | postgres
public | order_materials | таблица | postgres
public | order_services | таблица | postgres
public | orders        | таблица | postgres
public | payments      | таблица | postgres
public | reports       | таблица | postgres
public | secret_data   | таблица | postgres
public | services      | таблица | postgres
public | users         | таблица | postgres
(12 строк)

repair_workshop=#
```

Рисунок 1 – Доступ к базе данных

### 1.2.2 Микросервис арі

Вся основная логика отражена в Приложении А. JWT формируется на основе почты пользователя.

Настроено:

- Вход и аутентификация с помощью JWT;
- Логирование с помощью logger;
- Взаимодействие с базой данных с помощью SQLAlchemy;
- Схемы и модели для базы данных.

#### Листинг 3 – Часть кода файла models.py

```
class User(Base):
    __tablename__ = "users"
    id = Column(Integer, primary_key=True, index=True)
    email = Column(String, unique=True, index=True)
    hashed_password = Column(String)

class Client(Base):
    __tablename__ = "clients"
    client_id = Column(Integer, primary_key=True, index=True)
    full_name = Column(String, nullable=False)
    phone = Column(String, nullable=False)
    email = Column(String, nullable=True)

class Order(Base):
    __tablename__ = "orders"
    order_id = Column(Integer, primary_key=True, index=True)
    creation_date = Column(Date, nullable=False)
    completion_date = Column(Date, nullable=True)
    status = Column(String, nullable=False)
    total_cost = Column(DECIMAL, nullable=True)
    client_id = Column(Integer, ForeignKey("clients.client_id"),
    nullable=False)
    employee_id = Column(Integer, ForeignKey("employees.employee_id"),
    nullable=True)

    client = relationship("Client")
    employee = relationship("Employee")
```

#### Листинг 4 – Часть кода schemas.py

```
class ClientResponse(BaseModel):
    client_id: int
    full_name: str
    phone: str
    email: Optional[str] = None

    class Config:
        orm_mode = True
        from_attributes=True

class OrderResponse(BaseModel):
    order_id: int
    creation_date: date
```

```

completion_date: Optional[date] = None
status: str
total_cost: Optional[Decimal] = None
client_id: int

class Config:
    orm_mode = True
    from_attributes=True

```

## Листинг 5 – Dockerfile микросервиса api

```

FROM python:3.12

WORKDIR /api/

COPY requirements.txt .

RUN pip install --no-cache-dir -r requirements.txt

COPY . .

EXPOSE 8000

CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]

```

### 1.2.3 Микросервис nginx

Настроено проксирование внутри контейнера и логирование.

#### Листинг 6 – default.conf

```

server {
    listen 80;

    location / {
        proxy_pass http://api:8000;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
    }

    access_log /var/log/nginx/access.log;
    error_log /var/log/nginx/error.log;
}

```

### 1.2.4 docker-compose

Настроено:

- Запуск сервиса api строго после запуска контейнера база данных;
- Мэппинг томов для всех микросервисов для моментального контроля изменений;
- Мэппинг портов;
- Первоначальная инициализация базы данных в папке db.

## Листинг 7 – docker-compose.yml

```
services:
  db:
    image: postgres:15
    container_name: postgres-ISDB-lab4
    environment:
      POSTGRES_USER: ${POSTGRES_USER}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
      POSTGRES_DB: ${POSTGRES_DB}
    ports:
      - "5433:5432"
    volumes:
      - db_data:/var/lib/postgresql/data
      - ./db:/docker-entrypoint-initdb.d
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U ${POSTGRES_USER} -d
${POSTGRES_DB}"]
      interval: 1s
      timeout: 5s
      retries: 10

  api:
    build: ./api
    container_name: fastapi-ISBD-lab4
    volumes:
      - ./api:/api
    environment:
      SERVER_ID: SERVER-1
      DATABASE_URL:
postgresql://${POSTGRES_USER}:${POSTGRES_PASSWORD}@db:5432/${POSTGRES_DB}
    }
    depends_on:
      db:
        condition: service_healthy

  nginx:
    image: nginx:latest
    container_name: nginx-ISDB-lab4
    ports:
      - "80:80"
    volumes:
      - ./nginx:/etc/nginx/conf.d
    depends_on:
      - api

volumes:
  db_data:
```

### 1.3 Взаимодействие с веб-приложением и базой данных

Запуск производится с помощью команды `docker-compose up`.

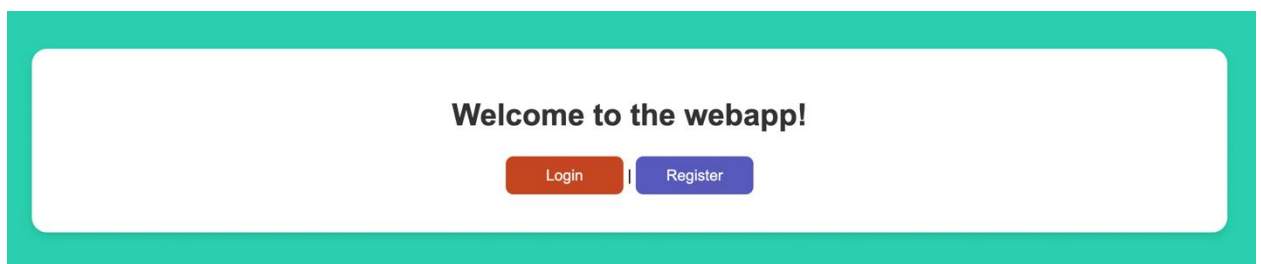
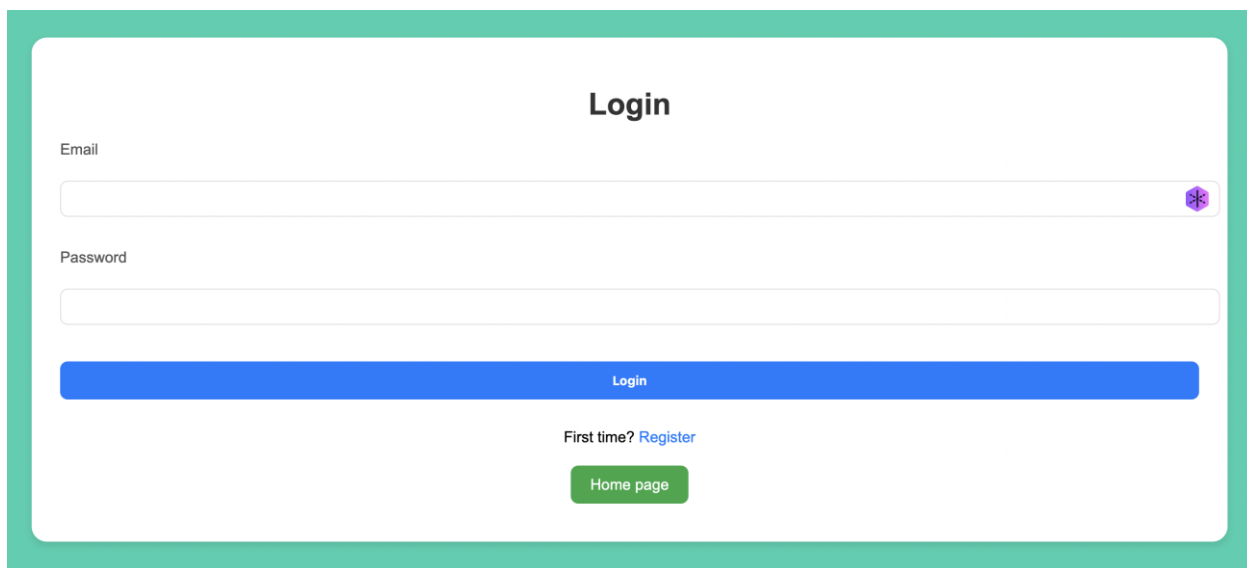


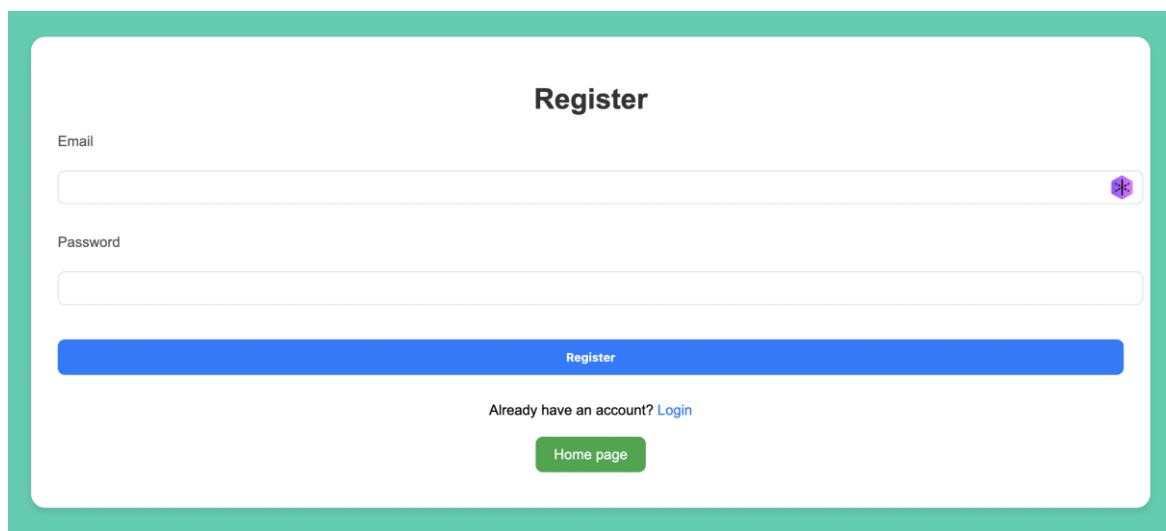
Рисунок 2 – Приветственная страница





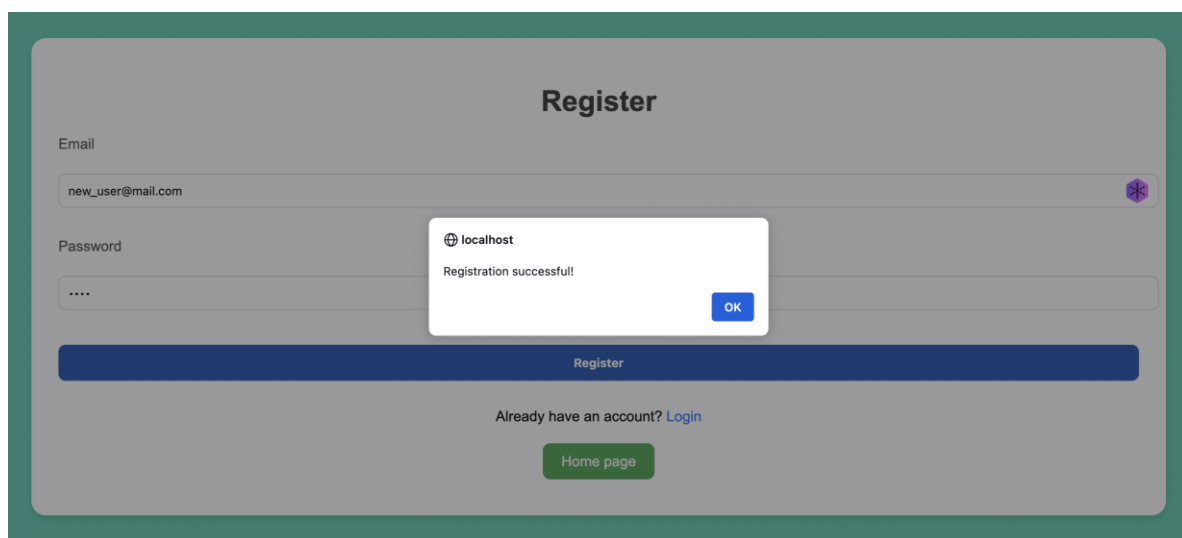
The image shows a login form titled "Login" centered at the top. Below the title, there are two input fields: "Email" and "Password". The "Email" field has a small purple icon with a star on its right side. Below the input fields is a large blue button labeled "Login". Underneath the button, there is a link "First time? Register" and a green button labeled "Home page".

Рисунок 3 – Страница входа



The image shows a registration form titled "Register" centered at the top. Below the title, there are two input fields: "Email" and "Password". The "Email" field has a small purple icon with a star on its right side. Below the input fields is a large blue button labeled "Register". Underneath the button, there is a link "Already have an account? Login" and a green button labeled "Home page".

Рисунок 4 – Страница регистрации



The image shows the same registration form as in Figure 4, but with a success message overlay. The "Email" field now contains the text "new\_user@mail.com". A white modal window is centered over the form, displaying a globe icon, the text "localhost", and "Registration successful!". There is a blue "OK" button in the bottom right corner of the modal. The "Register" button is now dark blue, and the "Home page" button is green.

Рисунок 5 – Окно после успешной регистрации

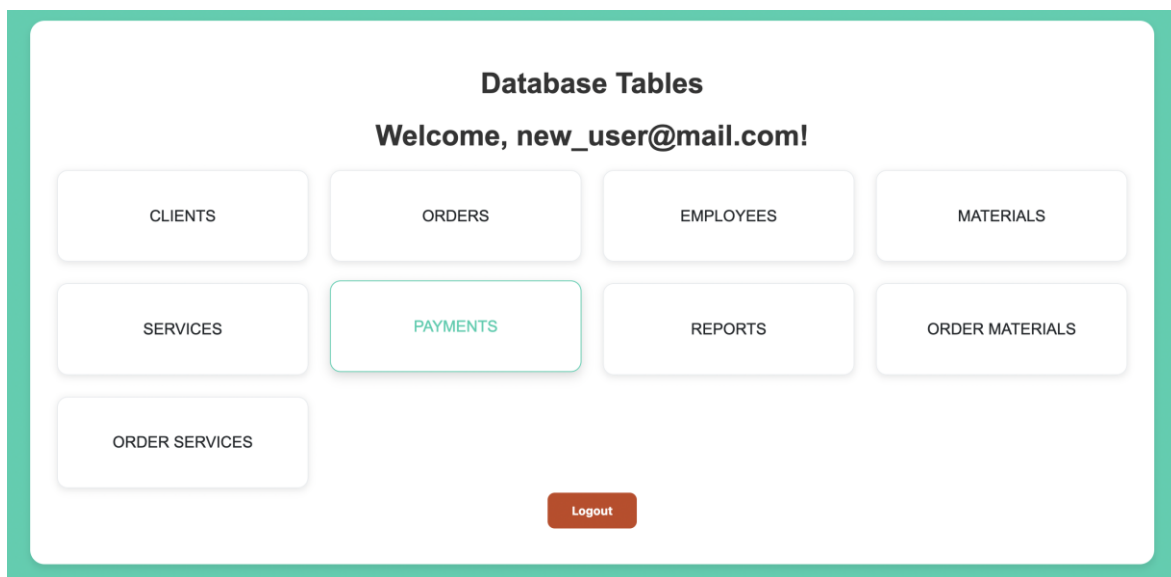


Рисунок 6 – Домашняя страница пользователя

Table: services			
service_id	name	base_cost	Actions
1	Ремонт компьютера	1000.00	<div>Edit</div> <div>Delete</div>
2	Замена жесткого диска	500.00	<div>Edit</div> <div>Delete</div>
3	Установка программного обеспечения	300.00	<div>Edit</div> <div>Delete</div>
4	Чистка ноутбука	700.00	<div>Edit</div> <div>Delete</div>
5	Диагностика	400.00	<div>Edit</div> <div>Delete</div>

Рисунок 7 – Таблица services

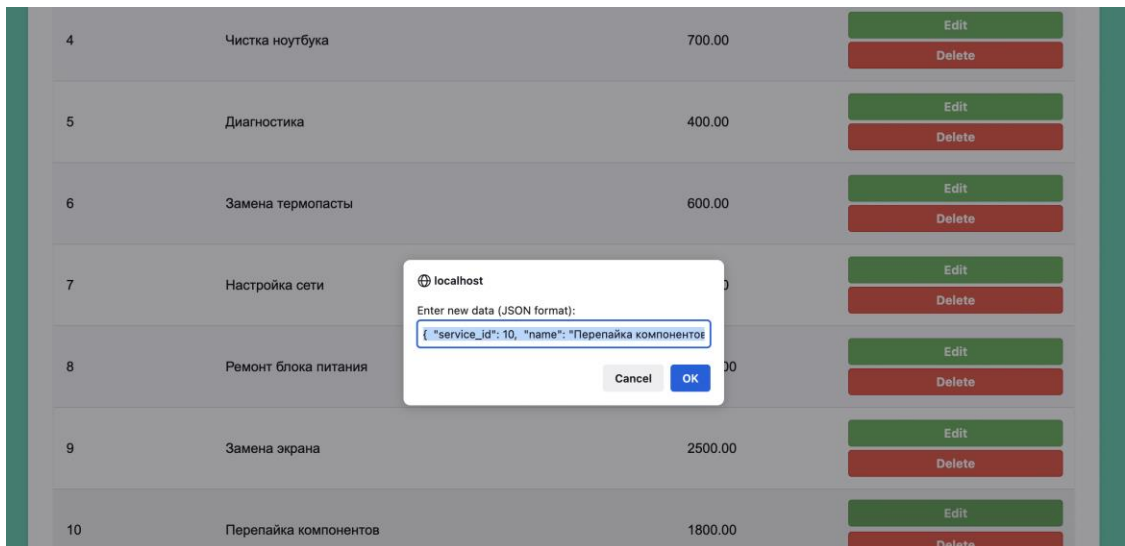


Рисунок 8 – Редактирование записи

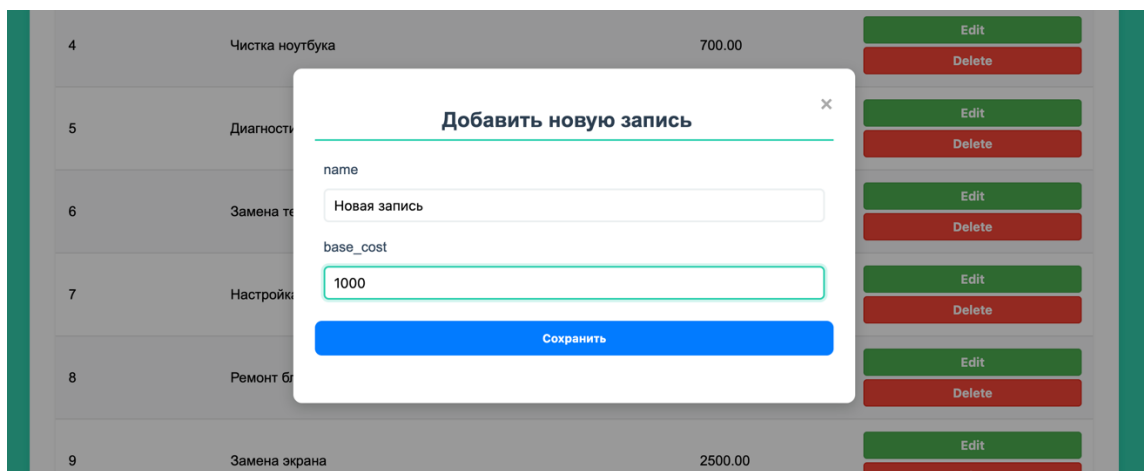


Рисунок 9 – Попытка добавления новой записи

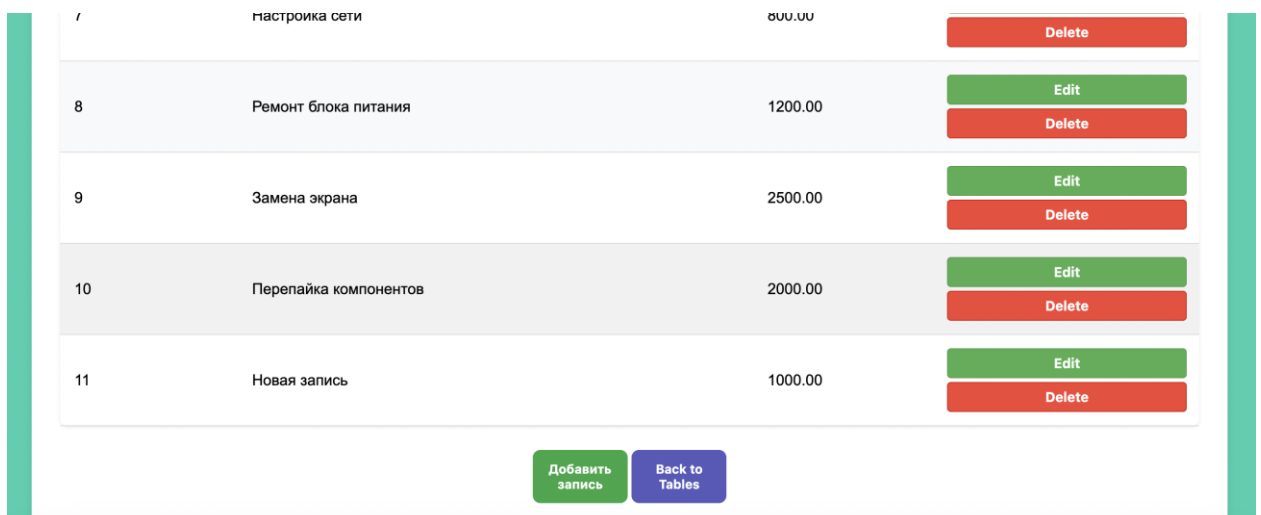


Рисунок 10 – Проверка добавления новой записи

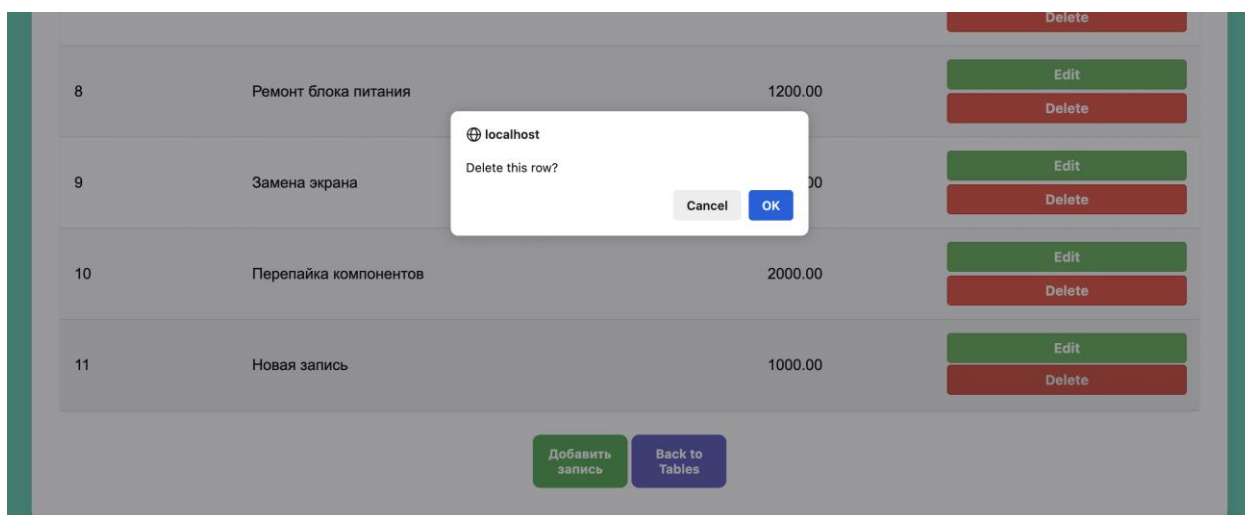


Рисунок 11 – Попытка удаление записи

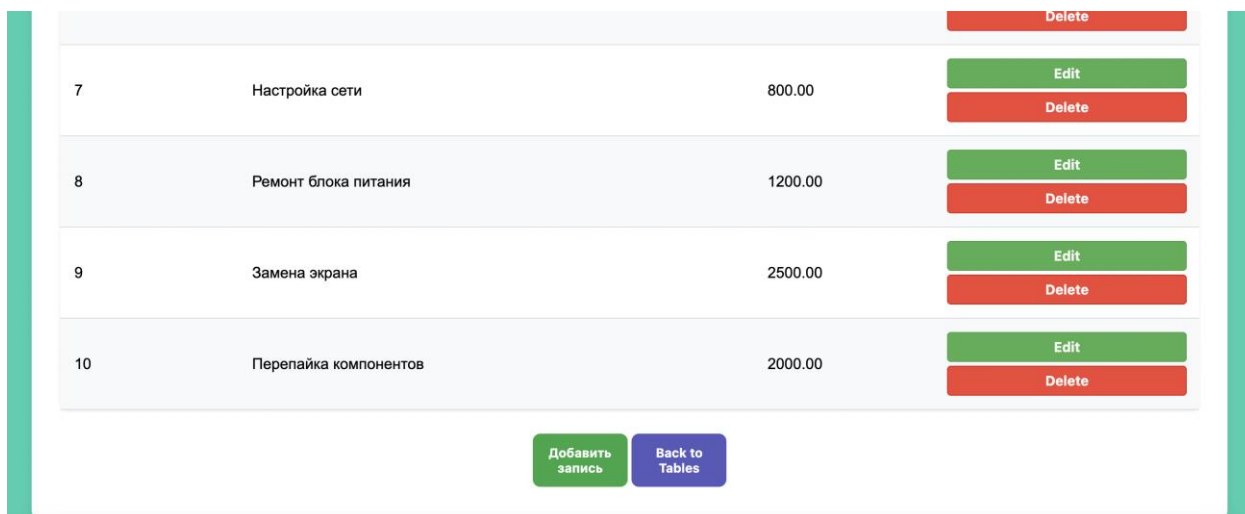


Рисунок 12 – Проверка удаления записи

## 1.4 Методы защиты данных

Аутентификация и авторизация:

- Используется JWT для доступа к веб-приложению. Также настроено управления сессиями пользователей.

Логирование действий:

- Все изменения в БД фиксируются в таблице-логе, что позволяет отслеживать действия пользователей.

Шифрование данных:

- Для хранения паролей пользователей используется хэширование с помощью библиотеки bcrypt.

## **ЗАКЛЮЧЕНИЕ**

В ходе выполнения лабораторной работы был разработан сервис для взаимодействия с базой данных ремонтной мастерской. Сервис поддерживает регистрацию и аутентификацию пользователей, просмотр, редактирование, удаление, добавление данных, а также логирование всех изменений в БД. Для обеспечения безопасности данных используются методы аутентификации на основе JWT, авторизации на основе JWT и шифрования.

## **СПИСОК ИСТОЧНИКОВ**

1. Документация FastAPI [Электронный ресурс]. – URL: <https://fastapi.tiangolo.com/> (Дата обращения: 25.01.2025).
2. Документация Jinja [Электронный ресурс]. – URL: <https://jinja.palletsprojects.com/en/stable/> (Дата обращения: 25.01.2025).

## ПРИЛОЖЕНИЕ А

### Листинг 8 – Программный код main.py (микросервис api)

```
from fastapi import FastAPI, Depends, HTTPException, status, Request, Query
# type: ignore
from fastapi.security import OAuth2PasswordRequestForm # type: ignore
from fastapi.responses import HTMLResponse, RedirectResponse # type: ignore
from fastapi.staticfiles import StaticFiles # type: ignore
from fastapi.templating import Jinja2Templates # type: ignore
from sqlalchemy.orm import Session, joinedload # type: ignore
from sqlalchemy.exc import SQLAlchemyError # type: ignore
from sqlalchemy import text # type: ignore
from fastapi.middleware.cors import CORSMiddleware # type: ignore
from typing import List, Optional, Dict, Any
from decimal import Decimal
import logging

from database import engine, Base, get_db
from schemas import UserCreate, Token, UserResponse
from utils import verify_password, get_password_hash
from auth import create_access_token, get_current_user
from models import User, Client, Order, Employee, Material, Service, Payment,
Report, OrderMaterial, OrderService
from schemas import ClientResponse, OrderResponse, EmployeeResponse,
MaterialResponse, ServiceResponse, PaymentResponse, ReportResponse,
OrderMaterialResponse, OrderServiceResponse, PaginatedResponse
import models

# create app
app = FastAPI()
app.mount("/static", StaticFiles(directory="static"), name="static")
templates = Jinja2Templates(directory="templates")

# create database tables
Base.metadata.create_all(bind=engine)

# create logger
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

# ===== CONST
# =====
PRIMARY_KEYS = {
    "clients": "client_id",
    "orders": "order_id",
    "employees": "employee_id",
    "materials": "material_id",
    "services": "service_id",
    "payments": "payment_id",
    "reports": "report_id",
    "order_materials": ["order_id", "material_id"],
    "order_services": ["order_id", "service_id"]
}
```

```

MODELS_MAPPING = {
    "clients": Client,
    "orders": Order,
    "employees": Employee,
    "materials": Material,
    "services": Service,
    "payments": Payment,
    "reports": Report,
    "order_materials": OrderMaterial,
    "order_services": OrderService,
}

SCHEMAS_MAPPING = {
    "clients": ClientResponse,
    "orders": OrderResponse,
    "employees": EmployeeResponse,
    "materials": MaterialResponse,
    "services": ServiceResponse,
    "payments": PaymentResponse,
    "reports": ReportResponse,
    "order_materials": OrderMaterialResponse,
    "order_services": OrderServiceResponse,
}

# ===== LOGIN LOGIC
=====
@app.get("/login", response_class=HTMLResponse)
async def login_page(request: Request):
    return templates.TemplateResponse(request=request, name="login.html",
context={"request": request})

@app.post("/login/", response_model=Token)
def login(form_data: OAuth2PasswordRequestForm = Depends(), db: Session =
Depends(get_db)):
    user = db.query(User).filter(User.email == form_data.username).first()

    if not user or not verify_password(form_data.password,
user.hashed_password):
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Incorrect email or password",
            headers={"WWW-Authenticate": "Bearer"},
        )

    # generate JWT token for authentication
    access_token = create_access_token(data={"sub": user.email})
    return {"access_token": access_token, "token_type": "bearer"}

# ===== REGISTRATION LOGIC
=====
@app.get("/register", response_class=HTMLResponse)
async def register_page(request: Request):
    return templates.TemplateResponse(request=request, name="register.html",
context={"request": request})

@app.post("/register/", response_model=Token)
def register(user: UserCreate, db: Session = Depends(get_db)):
    # check if user already exists
    db_user = db.query(User).filter(User.email == user.email).first()
    if db_user:
        raise HTTPException(status_code=400, detail="Email already
registered")

```



```

# hash the password and create the user
hashed_password = get_password_hash(user.password)
new_user = User(email=user.email, hashed_password=hashed_password)
db.add(new_user)
db.commit()
db.refresh(new_user)

# generate JWT token
access_token = create_access_token(data={"sub": new_user.email})
return {"access_token": access_token, "token_type": "bearer"}

# ===== TABLES LOGIC
=====
@app.get("/tables", response_class=HTMLResponse)
async def tables_page(request: Request):
    return templates.TemplateResponse(request=request, name="tables.html",
context={"request": request})

@app.get("/tables/{table_name}", response_class=HTMLResponse)
async def table_page(request: Request, table_name: str):
    return templates.TemplateResponse("table.html", {"request": request,
"table_name": table_name})

# ===== API TABLES LOGIC
=====
@app.get("/api/tables")
def get_available_tables():
    return {
        "tables": [
            "clients", "orders", "employees", "materials",
            "services", "payments", "reports",
            "order_materials", "order_services"
        ]
    }

@app.get("/api/tables/{table_name}", response_model=PaginatedResponse)
def get_table_data(table_name: str, db: Session = Depends(get_db)):
    try:
        model = MODELS_MAPPING.get(table_name)
        if not model:
            raise HTTPException(status_code=404, detail="Table not found")
        if table_name == "clients":
            data = db.query(model).order_by(model.client_id).all()
        elif table_name == "orders":
            data = db.query(model).order_by(model.order_id).all()
        elif table_name == "employees":
            data = db.query(model).order_by(model.employee_id).all()
        elif table_name == "materials":
            data = db.query(model).order_by(model.material_id).all()
        elif table_name == "services":
            data = db.query(model).order_by(model.service_id).all()
        elif table_name == "payments":
            data = db.query(model).order_by(model.payment_id).all()
        else:
            data = db.query(model).all()
        total_count = db.query(model).count()
        schema = SCHEMAS_MAPPING[table_name]
        serialized_data = [schema.from_orm(item) for item in data]

        return {"data": serialized_data, "totalCount": total_count}

    except AttributeError:

```

```

        raise HTTPException(status_code=404, detail="Table not found")
    except Exception as e:
        logger.error(f"exception: {str(e)}")
        raise HTTPException(status_code=400, detail=str(e))

# ===== CREATE & UPDATE & DELETE =====
@app.post("/api/tables/{table_name}")
def add_row(table_name: str, data: dict, db: Session = Depends(get_db)):
    try:
        model = MODELS_MAPPING.get(table_name)
        if not model:
            raise HTTPException(status_code=404, detail="Table not found")

        new_row = model(**data)
        db.add(new_row)
        db.commit()
        db.refresh(new_row)
        return {"message": "Row added successfully"}
    except SQLAlchemyError as e:
        db.rollback()
        raise HTTPException(status_code=400, detail=str(e._message))
    except Exception as e:
        raise HTTPException(status_code=500, detail="Internal server error")

@app.patch("/api/tables/{table_name}/{row_id}")
def update_row(table_name: str, row_id: str, data: dict, db: Session =
Depends(get_db)):
    try:
        model = MODELS_MAPPING.get(table_name)
        if not model:
            raise HTTPException(status_code=404, detail="Table not found")

        pk = PRIMARY_KEYS.get(table_name)

        # Обработка составных ключей
        if isinstance(pk, list):
            key_parts = row_id.split('-')
            if len(key_parts) != len(pk):
                raise HTTPException(status_code=400, detail="Invalid
composite key")

            filters = [getattr(model, pk[i]) == key_parts[i] for i in
range(len(pk))]
        else:
            filters = [getattr(model, pk) == row_id]

        row = db.query(model).filter(*filters).first()

        if not row:
            raise HTTPException(status_code=404, detail="Row not found")

        for key, value in data.items():
            if hasattr(model, key):
                setattr(row, key, value)
            else:
                raise HTTPException(status_code=400, detail=f"Invalid field:
{key}")

        db.commit()
        return {"message": "Row updated successfully"}

    except SQLAlchemyError as e:

```

```

        db.rollback()
        raise HTTPException(status_code=400, detail=str(e))
    except Exception as e:
        raise HTTPException(status_code=500, detail="Internal server error")

@app.delete("/api/tables/{table_name}/{row_id}")
def delete_row(table_name: str, row_id: int, db: Session = Depends(get_db)):
    try:
        model = MODELS_MAPPING.get(table_name)
        if not model:
            raise HTTPException(status_code=404, detail="Table not found")

        pk = PRIMARY_KEYS.get(table_name)
        if not pk:
            raise HTTPException(status_code=400, detail="Invalid table")

        row = db.query(model).filter(getattr(model, pk) == row_id).first()
        if not row:
            raise HTTPException(status_code=404, detail="Row not found")

        db.delete(row)
        db.commit()
        return {"message": "Row deleted successfully"}
    except SQLAlchemyError as e:
        db.rollback()
        raise HTTPException(status_code=400, detail=str(e._message))
    except Exception as e:
        raise HTTPException(status_code=500, detail="Internal server error")

# ===== ANOTHER LOGIC
# =====
@app.get("/home", response_class=HTMLResponse)
async def read_root(request: Request):
    # return {"message": "Welcome to the FastAPI Auth Demo"}
    return templates.TemplateResponse(request=request, name="home.html",
    context={"request": request})

@app.get("/")
def redirect_to_home():
    # redirect
    return RedirectResponse(url="/home", status_code=301)

# Protected route to check JWT token validity
@app.get("/me", response_model=UserResponse)
def access_cabinet(current_user: User = Depends(get_current_user)):
    return {"message": f"Welcome to your cabinet, {current_user.email}!",
    "user": current_user}

```