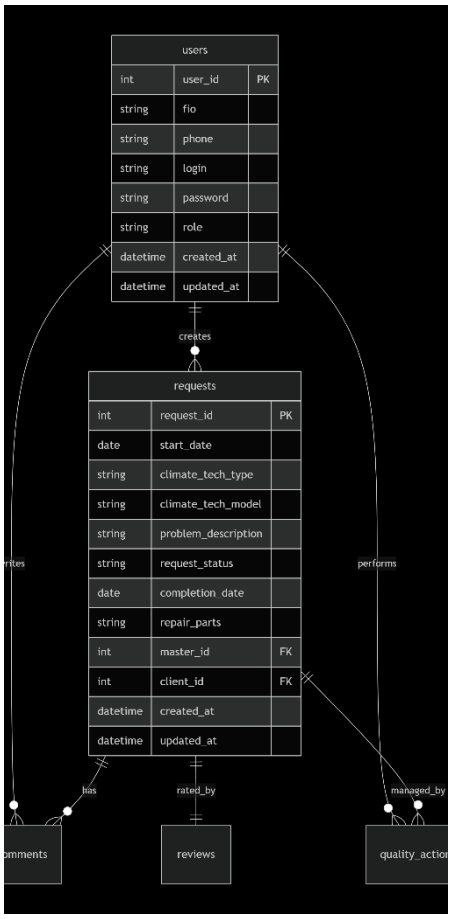
Отчет №2

1. **ER-диаграмма**

Спроектирована база данных в 3НФ с тремя основными сущностями. SQLAlchemy выбран как ORM для удобной работы с БД из Python.



2. **Схема БД в SQLite:**

Использованы внешние ключи с каскадным удалением для обеспечения ссылочной целостности данных. Все таблицы находятся в схеме по умолчанию.

Связи между таблицами:

- users → requests (client_id) - один клиент может иметь много заявок
- users → requests (master_id) - один специалист может выполнять много заявок
- users → comments - один пользователь может оставить много комментариев

- requests → comments - одна заявка может иметь много комментариев

3. Резервное копирование БД

Для SQLite реализовано два метода резервного копирования:

- Автоматическое резервное копирование (Python скрипт)

```
- import sqlite3
- import shutil
- import os
- from datetime import datetime
- import schedule
- import time
-
- def backup_database():
-     """Функция создания резервной копии базы данных"""
-     source_db = "repair_requests.db"
-     backup_dir = "backups"
-
-     # Создаем директорию для бэкапов, если её нет
-     if not os.path.exists(backup_dir):
-         os.makedirs(backup_dir)
-
-     # Формируем имя файла с датой
-     timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
-     backup_file = os.path.join(backup_dir, f"repair_requests_backup_{timestamp}.d
  b")
-
-     try:
-         # Создаем соединение с базой для инициализации WAL режима
-         conn = sqlite3.connect(source_db)
-         conn.close()
-
-         # Копируем файл базы данных
-         shutil.copy2(source_db, backup_file)
-
-         print(f"[{datetime.now()}] Backup created: {backup_file}")
-
-         # Удаляем старые бэкапы (старше 30 дней)
-         delete_old_backups(backup_dir, days=30)
-
-         return True
```

```
-       except Exception as e:
-           print(f"[{datetime.now()}] Backup failed: {e}")
-           return False
-
-   def delete_old_backups(backup_dir, days=30):
-       """Удаление старых резервных копий"""
-       import time
-
-       current_time = time.time()
-       for filename in os.listdir(backup_dir):
-           filepath = os.path.join(backup_dir, filename)
-           if os.path.isfile(filepath) and filename.endswith('.db'):
-               file_time = os.path.getmtime(filepath)
-               if (current_time - file_time) > (days * 24 * 3600):
-                   os.remove(filepath)
-                   print(f"Deleted old backup: {filename}")
-
-   def export_to_sql():
-       """Экспорт базы данных в SQL формат"""
-       import sqlite3
-
-       source_db = "repair_requests.db"
-       backup_dir = "backups"
-       timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
-       sql_file = os.path.join(backup_dir, f"repair_requests_export_{timestamp}.sql"
   )
-
-       try:
-           conn = sqlite3.connect(source_db)
-
-           with open(sql_file, 'w', encoding='utf-8') as f:
-               # Пишем заголовок
-               f.write(f"-- SQL Export of repair_requests.db\n")
-               f.write(f"-- Export time: {datetime.now()}\n")
-               f.write(f"-- Generated by Repair System Backup Tool\n\n")
-
-               # Получаем все таблицы
-               cursor = conn.cursor()
-               cursor.execute("SELECT name FROM sqlite_master WHERE type='table';")
-               tables = cursor.fetchall()
-
-               # Экспортируем каждую таблицу
-               for table in tables:
-                   table_name = table[0]
```

```
-
-                 # Получаем структуру таблицы
-                 cursor.execute(f"SELECT sql FROM sqlite_master WHERE type='table'
    AND name='{table_name}';")
-                 create_statement = cursor.fetchone()[0]
-                 f.write(f"{create_statement};\n\n")
-
-                 # Получаем данные таблицы
-                 cursor.execute(f"SELECT * FROM {table_name};")
-                 rows = cursor.fetchall()
-
-                 if rows:
-                     # Получаем имена столбцов
-                     cursor.execute(f"PRAGMA table_info({table_name});")
-                     columns = [col[1] for col in cursor.fetchall()]
-
-                     # Генерируем INSERT statements
-                     for row in rows:
-                         values = []
-                         for value in row:
-                             if value is None:
-                                 values.append("NULL")
-                             elif isinstance(value, str):
-                                 # Экранируем кавычки
-                                 value = value.replace("'", "''")
-                                 values.append(f"'{value}'")
-                             elif isinstance(value, datetime):
-                                 values.append(f"'{value.strftime('%Y-%m-%d %H:%M:
    %S')}'")
-                             else:
-                                 values.append(str(value))
-
-                         insert_sql = f"INSERT INTO {table_name} ({', '.join(colum
    ns)}) VALUES ({', '.join(values)});"
-                         f.write(f"{insert_sql}\n")
-
-                 f.write("\n")
-
-         conn.close()
-         print(f"[{datetime.now()}] SQL export created: {sql_file}")
-         return True
-
-     except Exception as e:
-         print(f"[{datetime.now()}] SQL export failed: {e}")
```

```
-           return False
-
-   # Запуск резервного копирования по расписанию
-   def schedule_backups():
-       """Настройка расписания резервного копирования"""
-       # Ежедневное резервное копирование в 2:00 ночи
-       schedule.every().day.at("02:00").do(backup_database)
-
-       # Еженедельный экспорт в SQL в воскресенье в 3:00
-       schedule.every().sunday.at("03:00").do(export_to_sql)
-
-       print("Backup scheduler started...")
-       while True:
-           schedule.run_pending()
-           time.sleep(60)
-
-   # Мгновенное создание резервной копии
-   def create_backup_now():
-       backup_database()
-       export_to_sql()
```

- Ручное резервное копирование через командную строку

```
-   # Windows
-   copy repair_requests.db repair_requests_backup_%date:~10,4%%date:~4,2%%date:~7,2%
    .db
-
-   # Linux/Mac
-   cp repair_requests.db repair_requests_backup_$(date +%Y%m%d_%H%M%S).db
```

- Экспорт данных в CSV

```
-   import sqlite3
-   import pandas as pd
-
-   def export_to_csv():
-       """Экспорт таблиц в CSV формат"""
-       conn = sqlite3.connect('repair_requests.db')
-
-       # Экспорт пользователей
-       users_df = pd.read_sql_query("SELECT * FROM users", conn)
-       users_df.to_csv('export_users.csv', index=False, encoding='utf-8')
-
-       # Экспорт заявок
```

```
-       requests_df = pd.read_sql_query("SELECT * FROM requests", conn)
-       requests_df.to_csv('export_requests.csv', index=False, encoding='utf-8')
-
-       # Экспорт комментариев
-       comments_df = pd.read_sql_query("SELECT * FROM comments", conn)
-       comments_df.to_csv('export_comments.csv', index=False, encoding='utf-8')
-
-       conn.close()
```

- Восстановление из резервной копии

```
-   def restore_from_backup(backup_file):
-       """Восстановление базы данных из резервной копии"""
-       import shutil
-       import os
-
-       try:
-           # Проверяем существование резервной копии
-           if not os.path.exists(backup_file):
-               print(f"Backup file {backup_file} not found")
-               return False
-
-           # Останавливаем приложение (если запущено)
-           # В реальном приложении здесь должна быть логика остановки соединений
-
-           # Копируем резервную копию на место основной базы
-           shutil.copy2(backup_file, 'repair_requests.db')
-
-           print(f"Database restored from {backup_file}")
-           return True
-
-       except Exception as e:
-           print(f"Restore failed: {e}")
-           return False
```

4. **Реализация ролей пользователей:**

Реализована система из 4 ролей с различными уровнями доступа:

```
# Модель управления ролями
ROLE_PERMISSIONS = {
    'Менеджер': {
        'can_view_all_requests': True,
```

```
        'can_create_request': True,
        'can_edit_request': True,
        'can_delete_request': True,
        'can_assign_master': True,
        'can_view_all_users': True,
        'can_create_user': True,
        'can_view_statistics': True,
        'can_view_comments': True,
        'can_add_comments': True,
        'can_change_status': True
    },
    'Оператор': {
        'can_view_all_requests': True,
        'can_create_request': True,
        'can_edit_request': True,
        'can_delete_request': False,
        'can_assign_master': False,
        'can_view_all_users': False,
        'can_create_user': False,
        'can_view_statistics': False,
        'can_view_comments': True,
        'can_add_comments': False,
        'can_change_status': True
    },
    'Специалист': {
        'can_view_all_requests': False,
        'can_create_request': False,
        'can_edit_request': True,
        'can_delete_request': False,
        'can_assign_master': False,
        'can_view_all_users': False,
        'can_create_user': False,
        'can_view_statistics': True,
        'can_view_comments': True,
        'can_add_comments': True,
        'can_change_status': True
    },
    'Заказчик': {
        'can_view_all_requests': False,
        'can_create_request': True,
        'can_edit_request': False,
        'can_delete_request': False,
        'can_assign_master': False,
        'can_view_all_users': False,
```

```
        'can_create_user': False,
        'can_view_statistics': False,
        'can_view_comments': False,
        'can_add_comments': False,
        'can_change_status': False
    }
}

# Функция проверки прав доступа
def check_permission(user_role, permission):
    """Проверка наличия прав у пользователя"""
    permissions = ROLE_PERMISSIONS.get(user_role, {})
    return permissions.get(permission, False)

# Middleware для проверки ролей
def require_permission(permission):
    """Декоратор для проверки разрешений"""
    from functools import wraps

    def decorator(func):
        @wraps(func)
        async def wrapper(*args, **kwargs):
            from fastapi import HTTPException, status
            from main import get_current_user

            # Получаем текущего пользователя
            current_user = await get_current_user()

            # Проверяем права
            if not check_permission(current_user.role, permission):
                raise HTTPException(
                    status_code=status.HTTP_403_FORBIDDEN,
                    detail=f"Permission '{permission}' required for this action"
                )

            return await func(*args, **kwargs)
        return wrapper
    return decorator
```

5. **Система аутентификации:**

Реализована JWT-аутентификация с использованием алгоритма HS256:

```python
from datetime import datetime, timedelta
from typing import Optional
from jose import JWTError, jwt
from passlib.context import CryptContext
from fastapi import HTTPException, status, Depends
from fastapi.security import OAuth2PasswordBearer

# Конфигурация JWT
SECRET_KEY = "your-secret-key-change-in-production-for-security"
ALGORITHM = "HS256"
ACCESS_TOKEN_EXPIRE_MINUTES = 60 * 24  # 24 часа

# Контекст для хеширования паролей
pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")
oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")

def verify_password(plain_password: str, hashed_password: str) -> bool:
    """Проверка пароля"""
    return pwd_context.verify(plain_password, hashed_password)

def get_password_hash(password: str) -> str:
    """Хеширование пароля"""
    return pwd_context.hash(password)

def create_access_token(data: dict, expires_delta: Optional[timedelta] = None):
    """Создание JWT токена"""
    to_encode = data.copy()

    if expires_delta:
        expire = datetime.utcnow() + expires_delta
    else:
        expire = datetime.utcnow() + timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)

    to_encode.update({"exp": expire})
    encoded_jwt = jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)
    return encoded_jwt

def decode_access_token(token: str):
    """Декодирование JWT токена"""
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
```

```
        return payload
    except JWTError:
        return None

# Middleware для защиты эндпоинтов
async def get_current_user(token: str = Depends(oauth2_scheme)):
    """Получение текущего пользователя из токена"""
    credentials_exception = HTTPException(
        status_code=status.HTTP_401_UNAUTHORIZED,
        detail="Could not validate credentials",
        headers={"WWW-Authenticate": "Bearer"},
    )

    try:
        payload = decode_access_token(token)
        if payload is None:
            raise credentials_exception

        user_id: int = payload.get("user_id")
        role: str = payload.get("role")
        fio: str = payload.get("fio")

        if user_id is None or role is None:
            raise credentials_exception

        # Здесь должна быть проверка пользователя в базе данных
        # Для упрощения возвращаем данные из токена
        return {
            "user_id": user_id,
            "role": role,
            "fio": fio
        }

    except JWTError:
        raise credentials_exception

# Эндпоинт аутентификации
from fastapi import APIRouter
from pydantic import BaseModel

router = APIRouter()

class LoginRequest(BaseModel):
    login: str
```

```python
    password: str

@router.post("/login")
async def login(request: LoginRequest):
    """Аутентификация пользователя"""
    # Здесь должна быть проверка логина и пароля в базе данных
    user = authenticate_user(request.login, request.password)

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

    # Создаем токен
    access_token = create_access_token(
        data={
            "user_id": user["user_id"],
            "role": user["role"],
            "fio": user["fio"]
        }
    )

    return {
        "access_token": access_token,
        "token_type": "bearer",
        "user": user
    }

def authenticate_user(login: str, password: str):
    """Аутентификация пользователя (заглушка)"""
    # В реальном приложении здесь будет запрос к базе данных
    return {
        "user_id": 1,
        "login": login,
        "role": "Менеджер",
        "fio": "Тестовый пользователь"
    }
```