Содержание

Задание 3
Введение 4
Выполнение работы 5
Клиентский интерфейс 5
Сущности 8
Технологический стек 9
Реализация 10
Заключение 11
Приложение 12
Приложение А «Код программы» 12
Приложение Б «Команды бота» 25

Задание

Система контроля доступа в помещения. Разработать и реализовать программные средства, выполняющие следующие задачи:

- удаленное администрирование:
  - создание, удаление, просмотр списка, настройка сущностей пользователя и администратора,
  - разрешение или запрет пользователям доступа в конкретные помещения,
- логирование посещений и возможность просмотра логов,
- удаленное управление дверьми (открытие | закрытие) помещений.

Введение

Целью данной работы является создание функционального и удобного для конечных пользователей и администраторов интерфейса управления системой контроля доступа в учебные помещения.

Под «системой контроля доступа» подразумевается программно-аппаратный комплекс, способный определять личность пользователя по изображению его лица (аутентификация) и, исходя из наличия «разрешения на доступ» в конкретное помещение, открывать или не открывать вход в помещение (авторизация); управляемый, настраиваемый и контролируемый администратором удаленно.

Выполнение работы

Клиентский интерфейс

Чтобы при взаимодействия с СКД администраторы и пользователи не были привязаны к какой-либо конкретной операционной системе (как, например, при использовании мобильного приложения), в роли клиентского интерфейса был выбран бот в мессенджере «Telegram». Имея открытое и развитое HTTP API для ботов (официальная полная документация]), выбранная платформа значительно упрощает разработку и позволяет не беспокоиться о различных тонкостях:

- согласованность интерфейса на различных платформах,
- простота разработки интерфейса (в сравнении с полноценными веб-приложениями)
- стабильность связи и надежность,
- обновления не требуют никаких дополнительных пользовательских манипуляций (в сравнении с мобильными приложениями).

Клиентский интерфейс можно разделить на разделы, включающие следующие команды (пример использования некоторых команд в приложении Б):

- Пользователи
  - Список всех добавленных пользователей
    `/users_list`
    Отвечает сообщением со списком всех пользователей с их ID.
  - Добавить нового пользователя
    `/new_user ((last_name)) ((first_name))` + прикреплен файл конфигурации лица (опционально)
    Отвечает сообщением с идентификатором нового пользователя.
  - Удалить пользователя
    `/delete_user ((user_id))`
    Отвечает сообщением со всеми данными пользователя, прикрепленным файлом конфигурации его лица.

- Настроить или обновить модель лица пользователя
  `/setup_user_model ((user_id))` + прикреплен файл конфигурации лица
  Отвечает сообщением со всеми данными пользователя и прикрепленным новым файлом конфигурации его лица.

- Помещения и доступ к ним
  - Список помещений с указанием их местоположения
    `/rooms_list`
    Отвечает сообщением со списком всех помещений в СКР и их местоположением.
  - Список пользователей, имеющих доступ к помещению
    `/allowed_users_list ((room_id))`
    Отвечает сообщением с ID, местоположением аудитории и списком пользователей (с их ID), которые имеют доступ к ней.
  - Дать пользователю доступ в аудиторию
    `/allow_access ((room_id)) ((user_id))`
    Отвечает сообщением с названием аудитории и списком пользователей, у которых имеется личный доступ.
  - Запретить пользователю доступ в аудиторию
    `/deny_access ((room_id)) ((user_id))`
    Отвечает сообщением с ID указанных пользователя и помещения.
  - Список администраторов аудитории
    `/room_admins_list ((room_id))`
    Отвечает сообщением с названием аудитории и списком администраторов с их никами в телеграм.
  - Дать администратору возможность «администрировать» аудиторию
    `/add_room_admin ((room_id)) ((admin_id))`
    Отвечает сообщением с названием аудитории и списком ее администраторов с их никами в телеграмм.
    Может выполнять только администратор указанной аудитории.
  - Запретить администратору «администрировать» аудиторию
    `/delete_room_admin ((room_id)) ((admin_id))`

Отвечает сообщением с названием аудитории и ID удаленного администратора с его ником в телеграмм.
Может выполнять только администратор указанной аудитории.

- Открыть вход в помещение на N-е время
`/open_door ((room_id)) ((time))`

- Администраторы
  - Список всех администраторов
`/admins_list`
Отвечает сообщением со списком всех администраторов с их никами в телеграмм.
  - Добавить нового администратора
`/new_admin @((telegram_username))`
Отвечает сообщением с ID и указанием, что администратор должен хоть раз написать боту, чтобы разрешить личные сообщения.
  - Удалить администратора
`/delete_admin @((telegram_username))`
Отвечает сообщением с ником удаленного администратора.

- Посещения
  - Список посещений аудитории за определенный день
`/room_visits_list ((room_id)) ((date))`
Отвечает сообщением со списком всех посещений в хронологическом порядке с указаением времени посещения.

  - Список посещений пользователя за определенный день
`/user_visits_list ((user_id)) ((date))`
Отвечает сообщением со списком всех посещений в хронологическом порядке с указаением времени посещения.

Изложенный выше пользовательский интерфейс в виде команд Telegram-боту является, буквально, MVP.

Сущности

Исходя из предыдущего пункта были выделены следующие сущности и их атрибуты, представленные на UML диаграмме на рисунке 1:

- User — объект пользователя, чей доступ контролируется,
- Admin — объект администратора,
- Room — объект помещения,
- Visit — объект записи о посещении помещения пользователем.

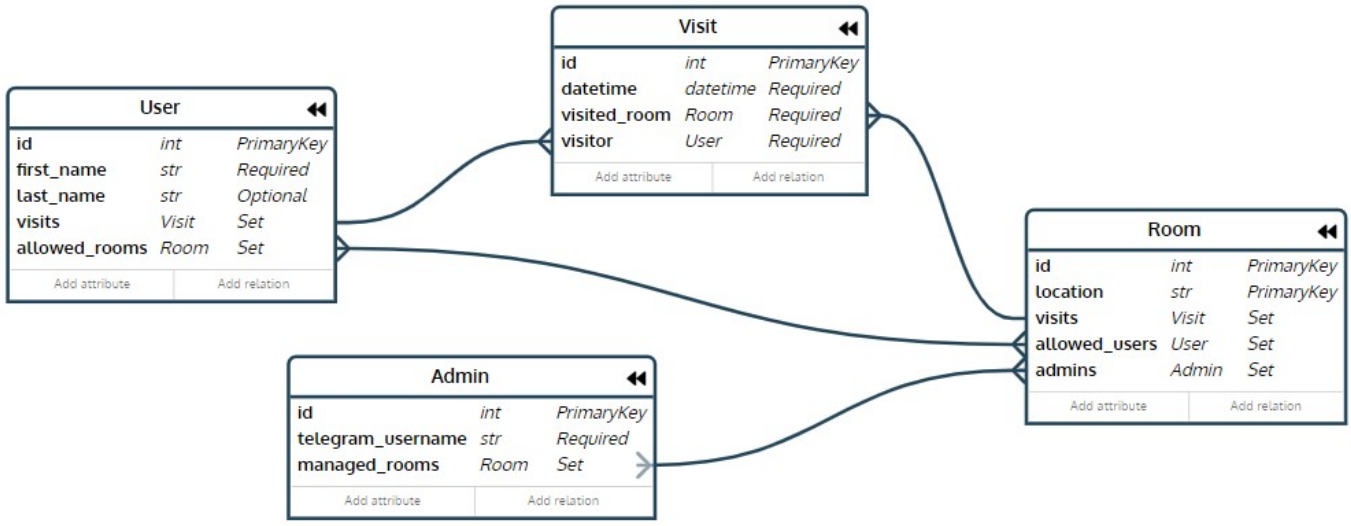

Рисунок 1 — UML-диаграмма выделенных программных сущностей

Технологический стек

- Для написания программы, реализующей необходимый функционал СКД, было принято решение использовать язык программирования «Python».
- Для упрощения взаимодействия с Telegram Bot API использовалась библиотека «pyTelegramBotAPI» (репозиторий на Github).
- Для взаимодействия с базой данных использовалась библиотека «Pony ORM» (официальный сайт), позволяющая отказаться от прямых SQL-запросов в БД в пользу стандартного синтаксиса Python.
- Для валидации параметров команд бота использовались регулярные выражения: CPython поставляется с модулем «re», имеющим все необходимые методы и функции для поиска и сравнения шаблонных строк.

Реализация

Был использован шаблон проектирования веб-приложений MVCS (Model – View – Controller — Service), предполагающий разделение программы на изолированные «слои» моделей (взаимодействия с БД), контроллеров (первичная обработка входящих запросов, валидация), представлений (подготовка данных для клиентов, в данном случае является частью контроллера) и сервисов (основная логика системы). Главная идея данного шаблона — инкапсуляция компонентов приложения. Подобное решение позволяет производить универсальный переиспользуемый код, при этом не сталкиваясь с потенциальными проблемами командной и (или) долгосрочной разработки.

Pony ORM предоставляет базовый мета-класс модели, наследники которого получают необходимые для манипуляции с БД методы и поля. Имплементация моделей сущностей произведена в файле «*entities.py*» (см. приложение А).

Получать входящие события (например, сообщений от пользователей) можно двумя методами:

- WebHooks – получение HTTP-запросов, содержащих новые события (инициатором является Telegram),
- LongPolling – запрос на открытие долгосрочного HTTP-соединения и ожидание новых событий (инициатором является сама программа-обработчик)

Т. к. для получения входящих HTTP-запросов требуется «белый» IPv4-адрес, более универсальным для встраиваемой системы будет LongPolling.

PyTelegramBotAPI предлагает добавлять обработчики входящих сообщений (и других событий) при помощи фабрик декораторов «*message_handler*» и подобных, являющихся атрибутами объектов класса «*Bot*». Оборачиваемая функция будет вызвана с одним неявным порядковым аргументом, являющимся объектом нового события (*Message* в случае обработчика сообщений). Имплементация обработчиков команд произведена в файлах «*user_controller.py*» , «*admin_controller.py*», «*room_controller.py*», «*visit_controller.py*» (см. приложение А).

Заключение

На данный момент, не смотря на то, что созданный интерфейс взаимодействия способен выполнять поставленные перед ним задачи, считать его готовым финальным решением пока нельзя. По моему мнению, в будущем требуется полностью отделить логику управления (в данном случае Telegram Bot) от бизнес-логики (сервисы), разработав и реализовав RESTful API, перейти от монолитного приложения к концепции «микросервисов».

В процессе выполнения предложенного задания приходилось сталкиваться с различными проблемами: от неочевидного поведения разметки *Markdown* при отправке сообщений до ошибок при работе с БД или регулярными выражениями. Разрешив все сложности, были улучшены навыки работы с *Python* и его прикладными библиотеками, углублены знания об устройстве и функционировании современных баз данных и т.д. Наиболее полезным приобретенным навыком, по моему мнению, является умение работать с регулярными выражениями.

Приложение

Приложение А «Код программы»

< entities.py >

```
from datetime import datetime

from pony.orm import *


db = Database()

sql_debug(True)

class User(db.Entity):
    """Пользователь системы контроля доступа"""
    id = PrimaryKey(int, auto=True)
    first_name = Required(str, 75)  # Имя пользователя
    last_name = Optional(str, 75)  # Фамилия пользователя
    visits = Set('Visit')  # Посещения пользователя
    allowed_rooms = Set('Room')  # Помещения, в которые разрешен доступ пользователю


class Room(db.Entity):
    """Помещение, в котором контролируется доступ"""
    id = PrimaryKey(int, auto=True)
    location = Required(str)  # Местоположение помещения
    visits = Set('Visit')  # Посещения помещения
    allowed_users = Set(User)  # Пользователи, которым разрешен доступ к помещению
    admins = Set('Admin')  # Администраторы, имеющие право управлять доступом в данном помещении


class Visit(db.Entity):
    """Посещение пользователем помещения в конкретное время"""
    id = PrimaryKey(int, auto=True)
    datetime = Required(datetime)  # Время посещения
    visited_room = Required(Room)  # Помещение, которое посетили
    visitor = Required(User)  # Посетитель


class Admin(db.Entity):
    """Администратор помещений"""
    id = PrimaryKey(int, auto=True)
    telegram_username = Required(str)  # имя пользователя администратора в телеграм
    managed_rooms = Set(Room)  # Помещения, доступом в которые имеет право управлять данный администрат
```

```
if __name__ == '__main__':
    db.bind(provider='sqlite',
            filename='C:/Users/markerviktor/Desktop/tpu_room_access_via_telegram/db.sqlite',
            create_db=True)
    db.generate_mapping(create_tables=True)
```

$< controllers/user_controller.py >$

```
import re
from typing import Tuple

from telebot import types

from room_access.app import bot
from room_access.services import user_service, exceptions
from room_access.controllers.utils import admin_required


@bot.message_handler(commands=['users_list'])
@admin_required
def users_list(message: types.Message):
    """Отвечает сообщением со списком всех пользователей и их ID"""
    users: Tuple[user_service.UserInfo] = user_service.get_all_users()

    answer_string = f"*Всего пользователей - {len(users)}:*\n" \
                    "`\{user\_id\} : \{last\_name\} \{first\_name\}`\n"

    for user in users:
        answer_string += f'{user.id} : {user.last_name} {user.first_name}\n'

    bot.send_message(chat_id=message.chat.id, text=answer_string, parse_mode='MarkdownV2')


@bot.message_handler(commands=['new_user'])
@admin_required
def new_user(message: types.Message):
    """Создает нового пользователя с указанными фамилией и именем"""

    # Имя или фамилия дожны начинаться с заглавной буквы,
    # содержать только кирилицу, иметь максимальную длину - 75 символов.
    # Пример команды: /new_user Маркер Виктор
    if not re.fullmatch(r'^/new_user [А-Я][а-я]{0,74} [А-Я][а-я]{0,74}$', message.text):
```

```
        bot.reply_to(message,
                     text='*Неверная команда\!*\n`\/new\_user \{first\_name\} \{last\_name\}`\n'
                          'Имя или фамилия нового пользователя дожны\n'
                          '- начинаться с __заглавной буквы__,\n'
                          '- содержать только __кирилицу__,\n'
                          '- иметь максимальную длину __75 символов__\.',
                     parse_mode='MarkdownV2')
        return None

    try:
        user = user_service.get_new_user(command_string=message.text)
        answer_text = 'Пользователь успешно создан.'
    except exceptions.AlreadyExist:
        answer_text = 'Пользователь с заданным сочетанием имени и фамилии уже существует!'

    bot.send_message(chat_id=message.chat.id, text=answer_text)


@bot.message_handler(commands=['delete_user'])
@admin_required
def delete_user(message: types.Message):
    """Удаляет пользователя по ID"""

    # ID пользователя может быть только числом.
    # Пример: /delete_user 12
    if not re.fullmatch(r'^/delete_user [0-9]+$', message.text):
        bot.reply_to(message, '*Неверная команда\!*\n`\/delete\_user \{user\_id\}`\n'
                              'ID пользователя можно узнать с помощью команды \/users\_list',
                     parse_mode='MarkdownV2')
        return None

    try:
        user_info = user_service.delete_user(command_string=message.text)
        answer_text = f'Удален пользователь:\n' \
                      f'{user_info.last_name} {user_info.first_name}'
    except exceptions.NotExist:
        answer_text = 'Пользователь с заданным ID не существует!'
    except exceptions.BadNumberOfArgs:
        answer_text = 'Неверное количество аргументов команды!'
    except exceptions.BadArgsTypes:
        answer_text = 'ID пользователя должно быть числом!'

    bot.send_message(chat_id=message.chat.id, text=answer_text)


@bot.message_handler(regexp=r"^/setup_user_model$")
@admin_required
def setup_user_model(message: types.Message):
```

```
    bot.send_message(chat_id=message.chat.id,
                     text='Функционал недоступен!')
```

$< controllers/admin_controller.py >$

```
import re

from telebot import types

from room_access.app import bot
from room_access.services import admin_service, exceptions
from room_access.controllers.utils import admin_required


@bot.message_handler(commands=['admins_list'])
def admins_list(message: types.Message):
    admins = admin_service.admins_list()

    answer_text = f"Всего администраторов - {len(admins)}\n" \
                  "`\{admin\_id\} : @username\n`"

    for admin in admins:
        answer_text += f'{admin.id} : @{admin.telegram_username}\n'

    bot.send_message(chat_id=message.chat.id, text=answer_text, parse_mode='MarkdownV2')


@bot.message_handler(commands=['new_admin'])
@admin_required
def new_admin(message: types.Message):
    """Добавляет нового администратора"""

    # Ник администратора должен начинаться с символа "@",
    # содержать только цифры и латинские буквы, иметь длину > 5 символов
    # Пример команды: /new_admin @MarkerViktor
    if not re.fullmatch(r'^/new_admin @[A-Za-z0-9]{5,32}$', message.text):
        bot.reply_to(message,
                     text='*Неверная команда\!*\n`\/new\_admin \@\{telegram\_username\}`\n'
                          'Ник должен\n'
                          '- начинаться с символа __\@__,\n'
                          '- содержать только __латиницу__, __цифры__ и __нижние подчеркивания,__\n'
                          '- иметь длину __от 5 до 32 символов__\.',
                     parse_mode='MarkdownV2')
        return None

    try:
```

```
        admin_service.new_admin(command_string=message.text)
        answer_text = 'Администратор успешно добавлен.'
    except exceptions.AlreadyExist:
        answer_text = 'Администратор с указанным ником уже существует!'
    except exceptions.BadNumberOfArgs:
        answer_text = 'Неверное количество аргументов команды!'
    except exceptions.BadArgs:
        answer_text = 'Неверный формат аргументов команды!'

    bot.send_message(chat_id=message.chat.id, text=answer_text)


@bot.message_handler(commands=['delete_admin'])
@admin_required
def delete_admin(message: types.Message):
    """Удаляет администратора"""

    # ID администратора может быть только числом.
    # Пример: /delete_admin 12
    if not re.fullmatch(r'^/delete_admin [0-9]+$', message.text):
        bot.reply_to(message, '*Неверная команда\!*\n`\/delete\_admin \{admin\_id\}`\n'
                              'ID администратора можно узнать с помощью команды \/admins\_list',
                     parse_mode='MarkdownV2')
        return None

    try:
        admin_service.delete_admin(command_string=message.text)
        answer_text = ''
    except exceptions.NotExist:
        pass

    bot.send_message(chat_id=message.chat.id, text=answer_text)
```

$< controllers/room_controller.py >$

```
import re
from typing import Tuple

from telebot import types

from room_access.app import bot
from room_access.services import room_service
from room_access.services import exceptions
from room_access.controllers.utils import admin_required
from room_access.services.entities_info import UserInfo
```

```
@bot.message_handler(commands=['rooms_list'])  # /rooms_list
def rooms_list(message: types.Message):
    """Отвечает сообщением со списком всех помещений и их ID"""
    rooms: Tuple[room_service.RoomInfo] = room_service.get_all_rooms()

    answer_string = f"*Всего помещений - {len(rooms)}:*\n" \
                    "`\{room\_id\} : \{room_location\}`\n"

    for room in rooms:
        answer_string += f'{room.id} : {room.location}\n'

    bot.send_message(chat_id=message.chat.id, text=answer_string, parse_mode='MarkdownV2')


@bot.message_handler(commands=['room_admins_list'])  # /room_admins_list {room_id}
def room_admins_list(message: types.Message):
    """Получить список администраторов помещения"""

    command_match = re.fullmatch(r"^(?P<command>/room_admins_list)\s(?P<room_id>\d+)$", message.text)
    if not command_match:
        bot.reply_to(message,
                     text='*Неверная команда\!*\n`\/room\_admins\_list \{room\_id\}`\n'
                          'ID помещения можно узнать с помощью команды \/rooms\_list\.',
                     parse_mode='MarkdownV2')
        return

    room_id = int(command_match['room_id'])

    try:
        room_admins = room_service.get_room_admins(room_id)
        answer_text = f"Администраторов помещения - {len(room_admins)}\n" \
                      "`\{admin\_id\} : @username\n`"
        for admin in room_admins:
            answer_text += f'{admin.id} : @{admin.telegram_username}\n'
    except exceptions.NotExist:
        answer_text = "Помещение с указанным ID не существует\."
    bot.send_message(chat_id=message.chat.id, text=answer_text, parse_mode='MarkdownV2')


@bot.message_handler(commands=['add_room_admin'])  # /add_room_admin {room_id} {admin_id}
@admin_required
def add_room_admin(message: types.Message):
    """Дает указанному администратору право управлением доступом в помещение"""

    # Регулярное выражение, соответствующее команде "/add_room_admin {room_id} {admin_id}",
    # разделенное на именованные группы command, room_id, admin_id
```

```
    command_match = re.fullmatch(r"^(?P<command>/add_room_admin)\s(?P<room_id>\d+)\s(?P<admin_id>\d+)$"
    if not command_match:
        bot.reply_to(message,
                     text='*Неверная команда\!*\n`\/add\_room\_admin \{room\_id\} \{admin\_id\}`\n'
                          'ID помещения можно узнать с помощью команды \/rooms\_list ,\n'
                          'ID администратора можно узнать с помощью команды \/admins\_list \.',
                     parse_mode='MarkdownV2')
        return

    room_id, admin_id = int(command_match['room_id']), int(command_match['admin_id'])

    # Проверка на то, что запрос от администратора указанного помещения
    if not room_service.check_room_admin(room_id, message.from_user.username):
        bot.send_message(chat_id=message.chat.id,
                         text='Вы не входите в список администраторов указанного помещения\n'
                              'Чтобы добавить администратора, нужно входить в список администраторов по
        return

    try:
        room_service.add_admin_to_room(room_id, admin_id)
        answer_string = "Администратор успешно добавлен\."
    except exceptions.NotExist:
        answer_string = "Для помещения или администратора указан неверный ID\."
    except exceptions.AlreadyExist:
        answer_string = "Указанный администратор уже входит в список администраторов указанного помещен

    bot.send_message(chat_id=message.chat.id, text=answer_string, parse_mode='MarkdownV2')


@bot.message_handler(commands=['delete_room_admin'])  # /delete_room_admin {room_id} {admin_id}
@admin_required
def delete_room_admin(message: types.Message):
    """Удаляет указанного администратора из списка администраторов указанного помещения"""

    # Регулярное выражение, соответствующее команде "/delete_room_admin {room_id} {admin_id}",
    # разделенное на именнованные группы command, room_id, admin_id
    command_match = re.fullmatch(r"^(?P<command>/delete_room_admin)\s(?P<room_id>\d)\s(?P<admin_id>\d)"
    if not command_match:
        bot.reply_to(message,
                     text='*Неверная команда\!*\n`\/delete\_room\_admin \{room\_id\} \{admin\_id\}`\n'
                          'ID помещения можно узнать с помощью команды \/rooms\_list ,\n'
                          'ID администратора можно узнать с помощью команды \/admins\_list \.',
                     parse_mode='MarkdownV2')
        return

    room_id, admin_id = int(command_match['room_id']), int(command_match['admin_id'])

    # Проверка на то, что запрос от администратора указанного помещения
```

```
    if not room_service.check_room_admin(room_id, message.from_user.username):
        bot.send_message(chat_id=message.chat.id,
                         text='Вы не входите в список администраторов указанного помещения.\n'
                              'Чтобы удалить администратора, нужно входить в список администраторов пом
        return

    try:
        room_service.delete_admin_from_room(room_id, admin_id)
        answer_string = "Администратор удален\."
    except exceptions.NotExist:
        answer_string = "Для помещения или администратора указан неверный ID\."

    bot.send_message(chat_id=message.chat.id, text=answer_string, parse_mode='MarkdownV2')


@bot.message_handler(commands=['allowed_users_list'])  # /allowed_users_list {room_id}
@admin_required
def allowed_users_list(message: types.Message):
    """Получить список пользователей, которым разрешен доступ в указанное помещение"""

    # Регулярное выражение, соответствующее команде "/allowed_users_list {room_id}",
    # разделенное на именнованные группы command, room_id
    command_match = re.fullmatch(r"^(?P<command>/allowed_users_list)\s(?P<room_id>\d)$", message.text)
    if not command_match:
        bot.reply_to(message,
                     text='*Неверная команда\!*\n`\/allowed\_users\_list \{room\_id\}`\n'
                          'ID помещения можно узнать с помощью команды \/rooms\_list\.',
                     parse_mode='MarkdownV2')
        return

    room_id = int(command_match['room_id'])

    try:
        allowed_users: Tuple[UserInfo, ...] = room_service.get_allowed_users(room_id)

        answer_string = f"*Разрешенных пользователей - {len(allowed_users)}:*\n" \
                        r"`\{user\_id\} : \{last\_name\} \{first\_name\}`\n"
        for user in allowed_users:
            answer_string += f'{user.id} : {user.last_name} {user.first_name}\n'

    except exceptions.NotExist:
        answer_string = r"Помещение с указанным ID не существует\."

    bot.send_message(chat_id=message.chat.id, text=answer_string, parse_mode='MarkdownV2')
```

$< services/user_service.py >$

```
from collections import namedtuple
from typing import Tuple

from room_access.services.entities_info import UserInfo
from room_access.utils import prepare_command_args
from room_access.services import exceptions as service_exceptions
from room_access.repositories import user_repository
from room_access.repositories import exceptions as repo_exceptions


def get_new_user(command_string: str) -> None:
    """Создание нового пользователя"""
    args: tuple = prepare_command_args(command_string)  # (last_name, first_name)
    try:
        last_name, first_name = args
    except ValueError:
        raise service_exceptions.BadNumberOfArgs()

    try:
        if user_repository.user_exist_by_first_and_last_name(first_name, last_name):
            raise service_exceptions.AlreadyExist()
        user_repository.create_user_entity(first_name, last_name)
    except repo_exceptions.BadValuesTypes:
        raise service_exceptions.BadArgsTypes()


def delete_user(command_string: str) -> UserInfo:
    """Удаление пользователя и получение информации о нем"""
    args: tuple = prepare_command_args(command_string)  # (user_id)
    try:
        user_id: int = args[0]
    except IndexError:
        raise service_exceptions.BadNumberOfArgs()

    try:
        user = user_repository.get_user_by_id(user_id)
    except repo_exceptions.EntityNotFound:
        raise service_exceptions.NotExist()
    user_info = UserInfo(None, user.first_name, user.last_name)

    user_repository.delete_user_by_id(user_id)
    return user_info


def get_all_users() -> Tuple[UserInfo]:
    """
    Получение информации обо всех пользователях
    в виде namedtuple с атрибутами id, first_name, last_name
```

```
    """
    users = user_repository.get_all_users()
    return tuple(map(lambda user: UserInfo(last_name=user.last_name,
                                           first_name=user.first_name,
                                           id=user.id), users))


if __name__ == '__main__':
    from room_access.entities import db

    db.bind(provider='sqlite', filename='/db.sqlite')
    db.generate_mapping()
```

$< services/admin_service.py >$

```
from typing import Tuple

from pony import orm

from room_access.services import exceptions as service_exceptions
from room_access.services.entities_info import AdminInfo
from room_access.utils import prepare_command_args
from room_access.repositories import admin_repository


def admins_list() -> Tuple[AdminInfo]:
    """
    Получение информации обо всех администраторах
    в виде namedtuple с атрибутами id, telegram_username
    """
    admins = admin_repository.get_all_admins()
    return tuple(map(lambda admin: AdminInfo(id=admin.id, telegram_username=admin.telegram_username), a


def new_admin(command_string: str):
    """Создание нового администратора"""
    args = prepare_command_args(command_string)  # (username)
    try:
        username: str = args[0]
    except IndexError:
        raise service_exceptions.BadNumberOfArgs()

    # проверка на символ "@" в начале имени пользователя
    if username.startswith('@'):
        username = username[1:]
    else:
```

```
        raise service_exceptions.BadArgs()

    if admin_repository.admin_exist_by_username(username):
        raise service_exceptions.AlreadyExist()

    admin_repository.create_admin_entity(username)


def delete_admin(command_string: str) -> AdminInfo:
    """Удаление администратора и возвращение информации о нем"""
    args = prepare_command_args(command_string)  # (admin_id)
    try:
        admin_id = args[0]
    except IndexError:
        raise service_exceptions.BadNumberOfArgs


if __name__ == '__main__':
    from room_access.entities import db, Admin, Room

    db.bind(provider='sqlite',
            filename='/db.sqlite')
    db.generate_mapping()
```

$< services/room_service.py >$

```
from collections import namedtuple
from typing import Tuple, NamedTuple

from pony import orm

from room_access.entities import Room, Admin
from room_access.services import exceptions
from room_access.services.entities_info import RoomInfo, AdminInfo, UserInfo


@orm.db_session
def get_all_rooms() -> Tuple[RoomInfo, ...]:
    """Получить кортеж с информацией о всех помещениях"""
    rooms = orm.select(room for room in Room)[:]
    return tuple(RoomInfo(room.id, room.location) for room in rooms)


@orm.db_session
def check_room_admin(room_id: int, telegram_username: str) -> bool:
    """Проверяет, входит ли указанный админимтратор в список администраторов указанной камнаты"""
```

```
    return Admin.get(telegram_username=telegram_username) in Room.get(id=room_id).admins


@orm.db_session
def get_room_admins(room_id: int) -> Tuple[AdminInfo, ...]:
    """Получить кортеж с информацией о всех администраторах помещения"""
    try:
        room: Room = Room[room_id]
    except orm.ObjectNotFound as e:
        raise exceptions.NotExist(e)
    return tuple(AdminInfo(admin.id, admin.telegram_username) for admin in room.admins)


@orm.db_session
def add_admin_to_room(room_id: int, admin_id: int) -> None:
    """Добавляет администратора указанному помещению"""
    try:
        admin: Admin = Admin[admin_id]
        room: Room = Room[room_id]
    except orm.ObjectNotFound as e:
        raise exceptions.NotExist(e)

    if admin in room.admins:
        raise exceptions.AlreadyExist("The admin has already in admins of the room")

    room.admins.add(admin)


@orm.db_session
def delete_admin_from_room(room_id: int, admin_id: int) -> None:
    """Удаляет администратора из указанного помещения"""
    try:
        admin: Admin = Admin[admin_id]
        room: Room = Room[room_id]
    except orm.ObjectNotFound as e:
        raise exceptions.NotExist(e)

    if admin not in room.admins:
        raise exceptions.BaseServiceException("The admin is not in admins of the room")

    room.admins.remove(admin)

@orm.db_session
def get_allowed_users(room_id) -> Tuple[UserInfo, ...]:
    """Получение кортежа с краткой информацией о пользователях, имеющих доступ в аудиторию"""
    try:
        room: Room = Room[room_id]
    except orm.ObjectNotFound as e:
```

```
        raise exceptions.NotExist(e)

    allowed_users = room.allowed_users
    return tuple(UserInfo(user.id, user.first_name, user.last_name) for user in allowed_users)
```

< app.py >

```
import logging
from concurrent.futures import ThreadPoolExecutor

from telebot import TeleBot

from room_access import config
from room_access import entities


# Настройка логирования
logging.basicConfig(
    #filename=config.LOG_PATH,
    level=logging.DEBUG,
)


# Инициализация сущностей бота
bot = TeleBot(
    token=config.TG_TOKEN,
    threaded=True,
    num_threads=2
)

# Импорт обработчиков команд
from room_access.controllers import user_controller
from room_access.controllers import help_controller
from room_access.controllers import admin_controller
from room_access.controllers import room_controller


# Инициализация БД
entities.db.bind(provider='sqlite', filename=config.DB_PATH)
entities.db.generate_mapping()
```

Приложение Б «Команды бота»

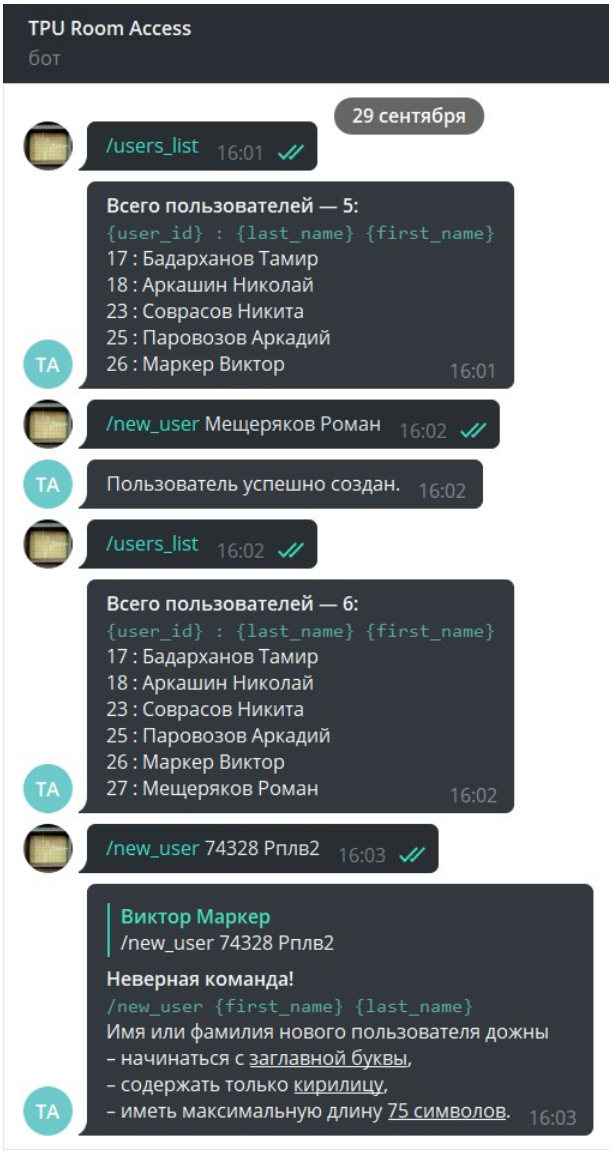

Рисунок 1 — Команды /users_list и /new_user

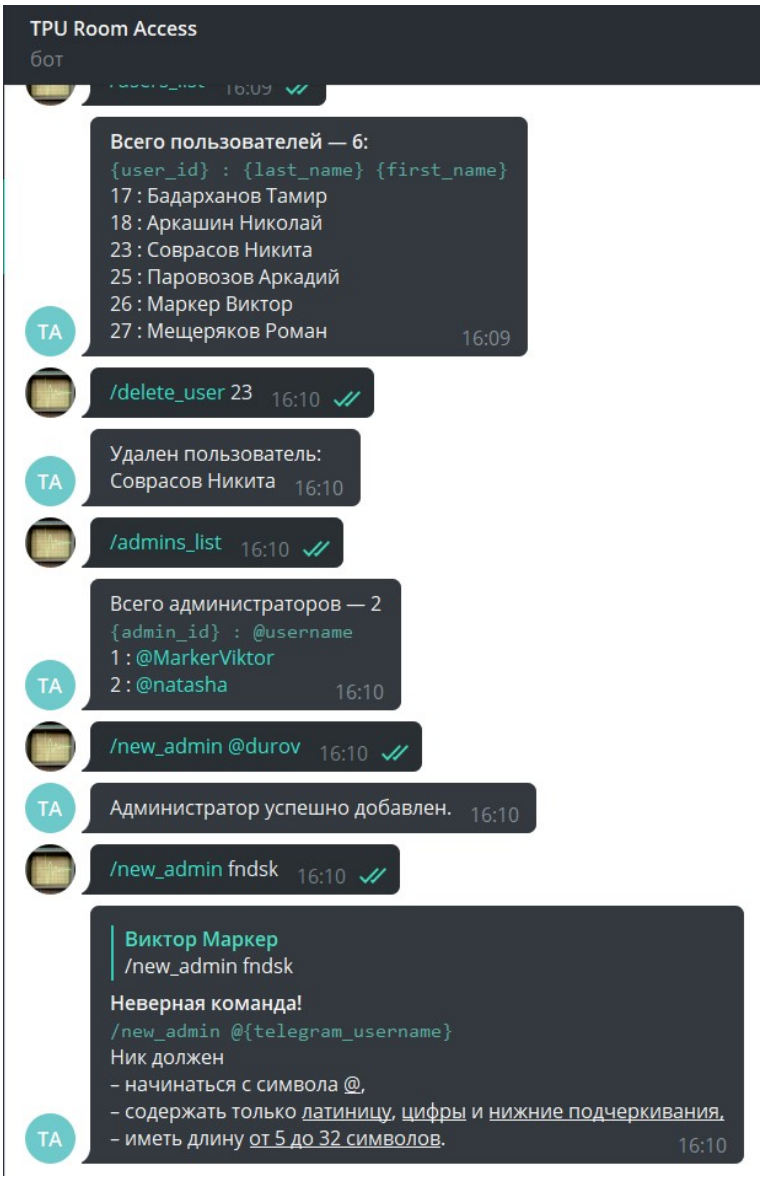

Рисунок 2 — Команды /delete_user, /admins_list и /new_admin

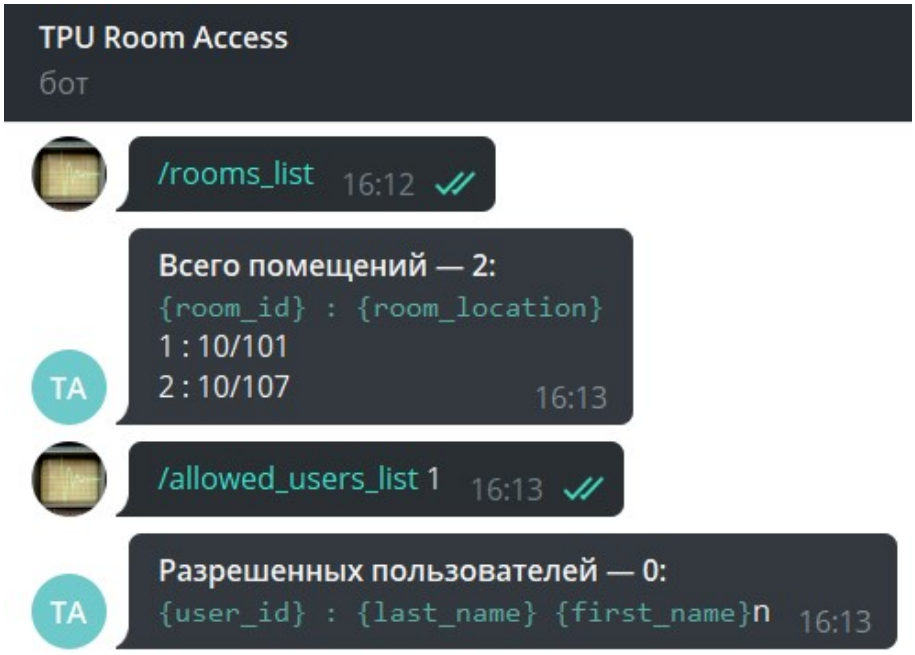

Рисунок 3 — Команды /rooms_list, /allowed_users_list, /room_admins_list

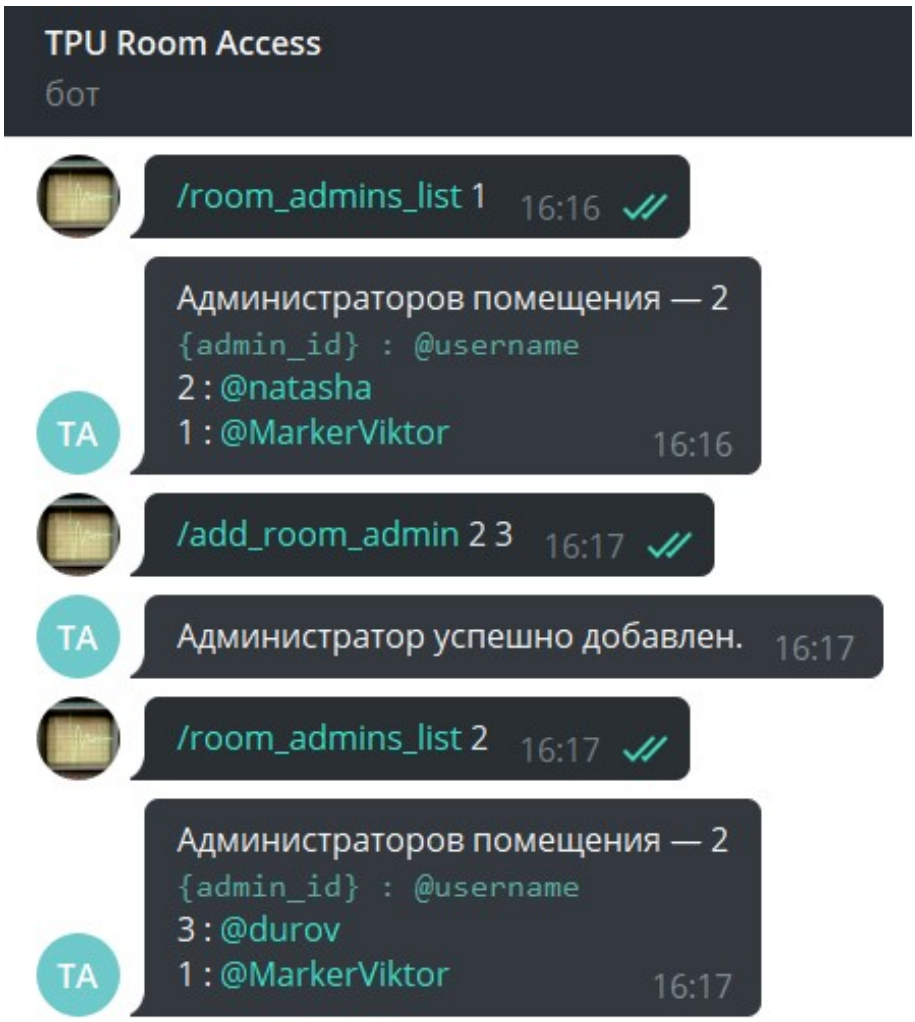

Рисунок 4 — Команды /room_admins_list и /add_room_admin