The Blooms Bridge software

Телеграм-бот «Автоматизация процесса записи клиентов в компанию по Личному бренду»

Отчет по разработке

«Beta release **v.1.0.1**»

Выдана: _____ Салимли Айзек Мухтар Оглы

Принято: _____ Демидова Елена Николаевна

«____»_____ 20___ г.

Санкт-Петербург, 2025

Содержание

Введение

В данном отчете:

- Схема бота
- Реализация бота
- Схема БД
- Реализация БД
- Схема парсера
- Реализация парсера
- Схема Maven проекта
- Реализация Maven проекта
- Реализация Conda проекта

1 Схема бота

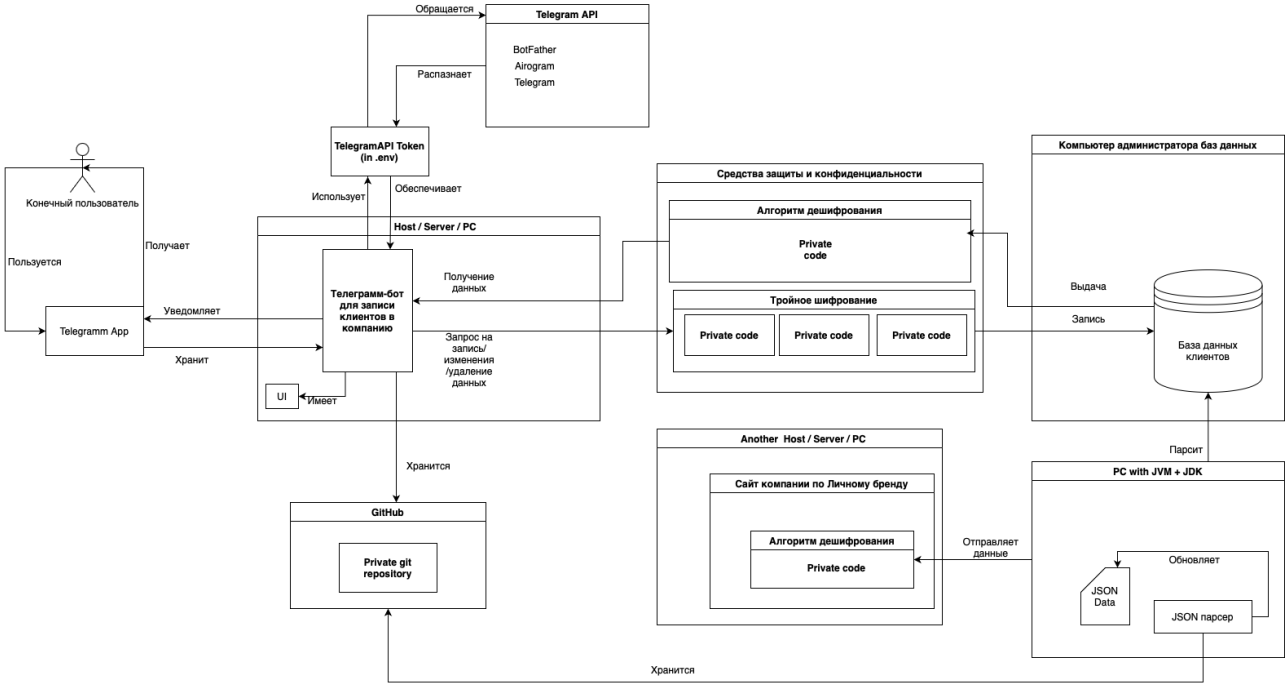

Рис. 1: Схема бота.

1.1 Чтение схемы бота

1. Телеграм-бот хранится на сервере/хосте/компьютере
2. Телеграм-бот хранится в репозитории на GitHub
3. Парсер хранится на компьютере (главного программиста), на ОС комьрютера обязательно установлен JVM + JDK
4. Конечный пользователь пользуется приложением Телеграм
5. Телеграм хранит в себе чат с ботом для записи в компанию по Личному бренду
6. Телеграм-бот имеет UI
7. Телеграм-бот использует .env файл в котором находится TelegramAPI Token
8. По токену бот обращается в TelegramAPI
9. TelegramAPI распознает Token
10. TelegramAPI обеспечивает бота с помощью Token
11. Бот может осуществлять запросы на запись, изменение, удаление данных. Данные проходят средства конфиденциальности
12. Средства конфиденциальности представляют собой три алгоритма шифрования (код скрыт) и средства дешифрования (код скрыт)
13. Запросы отправляются в базу данных клиентов
14. База данных хранится на компьютере администратора БД/ главного программиста
15. JSON парсер, парсит данные из базы данных.
16. Парсер создает .JSON файл, и отправляет его на сервера иного сайта (заказчика)

17. При выдачи данных из базы данных, данные проходят алгоритмы дешифрования (код скрыт)
18. Телеграм-бот получает дешифрованые данные
19. После получения данных, телеграм-бот уведомляет приложение
20. Уведомление получает конечный пользователь

2 Реализация бота

Управляющая логика бота, и его интерфейс, состоит из пяти файлов формата .py (Python файлы)

- UI.py - Файл с кодом графического интерфейса с использованием библиотек TelegramAPI
- main.py - Файл с кодом управляющей логики
- Encrypt.py - Файл с кодом алгоритмов шифрования (скрыт)
- Decrypt.py - Файл с кодом алгоритмов дешифрования (скрыт)
- db.py - Файл с кодом интегрированных запросов к базе данных

2.1 UI.py

```python
from telegram import InlineKeyboardButton, InlineKeyboardMarkup

def main_menu_keyboard():
    buttons = [
        [InlineKeyboardButton("Sign up", callback_data="sign_up")],
        [InlineKeyboardButton("Cancel", callback_data="cancel")],
        [InlineKeyboardButton("Change", callback_data="change")],
        [InlineKeyboardButton("Check out", callback_data="view")],
    ]
    return InlineKeyboardMarkup(buttons)

def build_time_keyboard(available_times, busy_slots, date_str, prefix="time_
    "):
    keyboard = []
    row = []
    for t in available_times:
        if (date_str, t) in busy_slots:
            continue
        row.append(InlineKeyboardButton(t, callback_data=f"{prefix}{t}"))
        if len(row) == 4:
            keyboard.append(row)
            row = []
    if row:
        keyboard.append(row)
    return InlineKeyboardMarkup(keyboard)
```

2.2 Encrypt.py

```
    from Crypto.Cipher import AES, DES3, ChaCha20
from Crypto.Random import get_random_bytes
from Crypto.Util.Padding import pad, unpad

def aes_encrypt(plaintext: bytes, key: bytes) -> bytes:
    cipher = AES.new(key, AES.MODE_CBC)
    ct_bytes = cipher.encrypt(pad(plaintext, AES.block_size))
    return cipher.iv + ct_bytes

def aes_decrypt(ciphertext: bytes, key: bytes) -> bytes:
    iv = ciphertext[:AES.block_size]
    ct = ciphertext[AES.block_size:]
    cipher = AES.new(key, AES.MODE_CBC, iv)
    return unpad(cipher.decrypt(ct), AES.block_size)

def des3_encrypt(plaintext: bytes, key: bytes) -> bytes:
    cipher = DES3.new(key, DES3.MODE_CBC)
    ct_bytes = cipher.encrypt(pad(plaintext, DES3.block_size))
    return cipher.iv + ct_bytes

def des3_decrypt(ciphertext: bytes, key: bytes) -> bytes:
    iv = ciphertext[:DES3.block_size]
    ct = ciphertext[DES3.block_size:]
    cipher = DES3.new(key, DES3.MODE_CBC, iv)
    return unpad(cipher.decrypt(ct), DES3.block_size)

def chacha20_encrypt(plaintext: bytes, key: bytes) -> bytes:
    cipher = ChaCha20.new(key=key)
    ciphertext = cipher.encrypt(plaintext)
    return cipher.nonce + ciphertext

def chacha20_decrypt(ciphertext: bytes, key: bytes) -> bytes:
    nonce = ciphertext[:8]
    ct = ciphertext[8:]
    cipher = ChaCha20.new(key=key, nonce=nonce)
    return cipher.decrypt(ct)
```

2.3 Decrypt.py

```
    from DB import get_connection
from Hashes import aes_decrypt

def get_clients():
    conn = get_connection()
    cursor = conn.cursor()
    cursor.execute("SELECT id, last_name, first_name, patronymic, phone FROM Client")
    rows = cursor.fetchall()
    cursor.close()
    conn.close()
    return rows

def main():
    clients = get_clients()
    for client in clients:
        client_id, last_name_enc, first_name_enc, patronymic_enc, phone_enc = client
        try:
            last_name = aes_decrypt(last_name_enc)
            first_name = aes_decrypt(first_name_enc)
            patronymic = aes_decrypt(patronymic_enc) if patronymic_enc else ""
            phone = aes_decrypt(phone_enc)
        except Exception as e:
            print(f"Error decrypting client {client_id}: {e}")
            continue
        print(f"Client {client_id}: {last_name} {first_name} {patronymic}, phone: {phone}")
```

2.4 db.py

```
    import mysql.connector
from mysql.connector import Error
from datetime import datetime

config = {
    'host': 'localhost',
    'user': 'root',
    'password': '***************',
    'database': 'booking_system'
}

def get_connection():
    try:
        connection = mysql.connector.connect(**config)
        if connection.is_connected():
            return connection
    except Error as e:
        print(f"Connection error: {e}")
        raise

def insert_log(log_text):
    try:
        conn = get_connection()
        cursor = conn.cursor()
        log_date = datetime.now().date()
        query = "INSERT INTO Logs (log_date, log_text) VALUES (%s, %s)"
        cursor.execute(query, (log_date, log_text))
        conn.commit()
        cursor.close()
        conn.close()
    except Error as e:
        print(f"Failed to insert log: {e}")

def insert_client(last_name, first_name, patronymic, phone):
    try:
        conn = get_connection()
        cursor = conn.cursor()
        query = """
            INSERT INTO Client (last_name, first_name, patronymic, phone)
            VALUES (%s, %s, %s, %s)
        """
        cursor.execute(query, (last_name, first_name, patronymic, phone))
        conn.commit()
        client_id = cursor.lastrowid
        print(f"insert_client: client_id = {client_id}")
        cursor.close()
        conn.close()
        return client_id
    except Error as e:
        insert_log(f"Error inserting client: {e}")
        raise

def insert_appointment(client_id, appointment_date, appointment_time,
    full_name):
    try:
        conn = get_connection()
        cursor = conn.cursor()
        query = """
```

```
58              INSERT INTO Appointment (client_id, appointment_date,
                    appointment_time, full_name)
59              VALUES (%s, %s, %s, %s)
60          """
61          cursor.execute(query, (client_id, appointment_date, appointment_time
                , full_name))
62          conn.commit()
63          appointment_id = cursor.lastrowid
64          print(f"insert_appointment: appointment_id = {appointment_id}")
65          cursor.close()
66          conn.close()
67          return appointment_id
68      except Error as e:
69          insert_log(f"Error inserting appointment: {e}")
70          raise
71
72  def update_appointment(appointment_id, appointment_date, appointment_time,
        full_name):
73      try:
74          conn = get_connection()
75          cursor = conn.cursor()
76          query = """
77              UPDATE Appointment
78              SET appointment_date = %s, appointment_time = %s, full_name = %s
79              WHERE id = %s
80          """
81          cursor.execute(query, (appointment_date, appointment_time, full_name
                , appointment_id))
82          conn.commit()
83          print(f"update_appointment: updated appointment_id = {appointment_id
                }")
84          cursor.close()
85          conn.close()
86      except Error as e:
87          insert_log(f"Error updating appointment {appointment_id}: {e}")
88          raise
89
90  def insert_status(appointment_id, status, client_phone, client_full_name):
91      try:
92          conn = get_connection()
93          cursor = conn.cursor()
94          query = """
95              INSERT INTO AppointmentStatus (appointment_id, status,
                    client_phone, client_full_name)
96              VALUES (%s, %s, %s, %s)
97          """
98          cursor.execute(query, (appointment_id, status, client_phone,
                client_full_name))
99          conn.commit()
100         status_id = cursor.lastrowid
101         print(f"insert_status: status_id = {status_id}")
102         cursor.close()
103         conn.close()
104         return status_id
105     except Error as e:
106         insert_log(f"Error inserting appointment status for appointment {
                appointment_id}: {e}")
107         raise
108
109 def update_status(appointment_id, status, client_phone, client_full_name):
110     try:
```

```
111         conn = get_connection()
112         cursor = conn.cursor()
113         query = """
114             UPDATE AppointmentStatus
115             SET status = %s, client_phone = %s, client_full_name = %s
116             WHERE appointment_id = %s
117         """
118         cursor.execute(query, (status, client_phone, client_full_name,
                appointment_id))
119         conn.commit()
120         print(f"update_status: updated appointment_id = {appointment_id}")
121         cursor.close()
122         conn.close()
123     except Error as e:
124         insert_log(f"Error updating status for appointment {appointment_id}:
                {e}")
125         raise
```

2.5 main.py

```
import logging
import re
import os
import threading
import base64
from telegram.ext import ContextTypes
from datetime import datetime, time, timedelta
from pytz import timezone
from dotenv import load_dotenv
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
import subprocess

from DB import insert_client, insert_appointment, insert_status,
    update_appointment, update_status, insert_log
from telegram import Update, InlineKeyboardButton, InlineKeyboardMarkup
from telegram.ext import (
    Application,
    CommandHandler,
    CallbackQueryHandler,
    MessageHandler,
    ConversationHandler,
    ContextTypes,
    filters
)
from UI import main_menu_keyboard, build_time_keyboard

load_dotenv()
TOKEN = os.getenv("TELEGRAM_API_TOKEN")
if not TOKEN:
    raise ValueError("Not found! TELEGRAM_API_TOKEN in .env file")

logging.basicConfig(
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
    level=logging.INFO
)
logger = logging.getLogger(__name__)

AES_KEY = b'mysecretkey12345'

def aes_encrypt(plaintext: str) -> str:
    data = plaintext.encode('utf-8')
    cipher = AES.new(AES_KEY, AES.MODE_CBC)
    ct_bytes = cipher.encrypt(pad(data, AES.block_size))
    encrypted = cipher.iv + ct_bytes
    return base64.b64encode(encrypted).decode('utf-8')

def aes_decrypt(ciphertext_b64: str) -> str:
    ciphertext = base64.b64decode(ciphertext_b64)
    iv = ciphertext[:AES.block_size]
    ct = ciphertext[AES.block_size:]
    cipher = AES.new(AES_KEY, AES.MODE_CBC, iv)
    data = unpad(cipher.decrypt(ct), AES.block_size)
    return data.decode('utf-8')

appointments = {}
busy_slots = {}  # busy_slots[(date, time)] = user_id

AVAILABLE_TIMES = []
```

```
    hour = 8
    minute = 0
    while True:
        h_str = str(hour).zfill(2)
        m_str = str(minute).zfill(2)
        tm = f"{h_str}:{m_str}"
        AVAILABLE_TIMES.append(tm)
        minute += 20
        if minute >= 60:
            hour += 1
            minute -= 60
        if hour > 20:
            break

    (
        STATE_MENU,
        STATE_SIGNUP_NAME,
        STATE_SIGNUP_PHONE,
        STATE_SIGNUP_DATE,
        STATE_SIGNUP_TIME,
        STATE_CANCEL_CONFIRM,
        STATE_CHANGE_NAME,
        STATE_CHANGE_DATE,
        STATE_CHANGE_TIME,
        STATE_NO_SHOW_REASON,
    ) = range(10)

    async def start_command(update: Update, context: ContextTypes.DEFAULT_TYPE):
        await update.message.reply_text(
            "Hello, I'm the appointment bot!\n\nSend /start to open the menu and make an appointment."
        )

    async def info_command(update: Update, context: ContextTypes.DEFAULT_TYPE):
        await update.message.reply_text(
            "Welcome! Choose an action:",
            reply_markup=main_menu_keyboard()
        )
        return STATE_MENU
    async def menu_callback(update: Update, context: ContextTypes.DEFAULT_TYPE):
        query = update.callback_query
        user_id = query.from_user.id
        await query.answer()
        if query.data == "sign_up":
            if user_id in appointments:
                decrypted_name = aes_decrypt(appointments[user_id]['name'])
                await query.message.reply_text(
                    f"You already have an appointment ({decrypted_name}). Please cancel or change it first."
                )
                await query.message.reply_text("Returning you to the menu.", reply_markup=main_menu_keyboard())
                return STATE_MENU
            else:
                await query.message.reply_text("Enter your full name (at least 2 words):")
                return STATE_SIGNUP_NAME
        elif query.data == "cancel":
```

```
113             if user_id not in appointments:
114                 await query.message.reply_text("You have no appointments to
                        cancel.")
115                 await query.message.reply_text("Returning you to the menu.",
                        reply_markup=main_menu_keyboard())
116                 return STATE_MENU
117             else:
118                 info = appointments[user_id]
119                 decrypted_name = aes_decrypt(info['name'])
120                 decrypted_phone = aes_decrypt(info['phone'])
121                 await query.message.reply_text(
122                     f"Are you sure you want to cancel your appointment:\n{
                            decrypted_name} ({decrypted_phone}) on {info['date']}
                            at {info['time']}?\nReply 'Yes' or 'No'."
123                 )
124                 return STATE_CANCEL_CONFIRM
125         elif query.data == "change":
126             if user_id not in appointments:
127                 await query.message.reply_text("You have no appointments to
                        change.")
128                 await query.message.reply_text("Returning you to the menu.",
                        reply_markup=main_menu_keyboard())
129                 return STATE_MENU
130             else:
131                 decrypted_name = aes_decrypt(appointments[user_id]['name'])
132                 await query.message.reply_text(
133                     f"Current name: {decrypted_name}\n\nEnter new name or
                            '-' to skip:"
134                 )
135                 return STATE_CHANGE_NAME
136         elif query.data == "view":
137             if user_id not in appointments:
138                 await query.message.reply_text("You have no appointments.")
139             else:
140                 info = appointments[user_id]
141                 decrypted_name = aes_decrypt(info['name'])
142                 decrypted_phone = aes_decrypt(info['phone'])
143                 text = f"Your appointment:\nName: {decrypted_name}\nPhone: {
                        decrypted_phone}\nDate: {info['date']}\nTime: {info['time
                        ']}"
144                 await query.message.reply_text(text)
145             await query.message.reply_text("Returning you to the menu.",
                    reply_markup=main_menu_keyboard())
146             return STATE_MENU
147         else:
148             await query.message.reply_text("Unknown command. Returning you
                    to the menu.", reply_markup=main_menu_keyboard())
149             return STATE_MENU
150             async def sign_up_name(update: Update, context: ContextTypes.
                    DEFAULT_TYPE):
151             name = update.message.text.strip()
152             parts = name.split()
153             if len(parts) < 2:
154                 await update.message.reply_text("Oops! Your name must
                        contain at least 2 words. Appointment cancelled.")
155                 await update.message.reply_text("Returning you to the menu."
                        , reply_markup=main_menu_keyboard())
156                 return STATE_MENU
157             context.user_data['name'] = name
158             await update.message.reply_text("Enter your phone number in the
                    format:\n+7 XXX XXX XX XX or 8 XXX XXX XX XX")
```

```
159             return STATE_SIGNUP_PHONE
160
161     async def sign_up_phone(update: Update, context: ContextTypes.
            DEFAULT_TYPE):
162             phone = update.message.text.strip()
163             pattern = r"^(?:\+7|8)\s?\d{3}\s?\d{3}\s?\d{2}\s?\d{2}$"
164             if not re.match(pattern, phone):
165                 await update.message.reply_text("Invalid phone format!
                        Appointment cancelled.")
166                 await update.message.reply_text("Returning you to the menu."
                        , reply_markup=main_menu_keyboard())
167                 return STATE_MENU
168             context.user_data['phone'] = phone
169             await update.message.reply_text("Enter the date (e.g., YYYY-MM-
                    DD or DD.MM.YYYY):")
170             return STATE_SIGNUP_DATE
171             async def sign_up_date(update: Update, context: ContextTypes.
                    DEFAULT_TYPE):
172             date_str = update.message.text.strip()
173             parsed_date = None
174             for fmt in ["%Y-%m-%d", "%d.%m.%Y"]:
175                 try:
176                     parsed_date = datetime.strptime(date_str, fmt)
177                     break
178                 except ValueError:
179                     pass
180             if not parsed_date:
181                 await update.message.reply_text("Unrecognized date format.
                        Please re-enter (YYYY-MM-DD or DD.MM.YYYY).");
182                 return STATE_SIGNUP_DATE
183             today = datetime.now().date()
184             if parsed_date.date() < today:
185                 await update.message.reply_text("Oops! This date has already
                         passed. Appointment cancelled.")
186                 await update.message.reply_text("Returning you to the menu."
                        , reply_markup=main_menu_keyboard())
187                 return STATE_MENU
188             normalized_date_str = parsed_date.strftime("%Y-%m-%d")
189             context.user_data['date'] = normalized_date_str
190             now = datetime.now()
191             if parsed_date.date() == today:
192                 current_time = now.time()
193                 filtered_times = [t for t in AVAILABLE_TIMES if time(*map(
                        int, t.split(":"))) > current_time]
194             else:
195                 filtered_times = AVAILABLE_TIMES
196             if not filtered_times:
197                 await update.message.reply_text("No available slots today.
                        Appointment cancelled.")
198                 await update.message.reply_text("Returning you to the menu."
                        , reply_markup=main_menu_keyboard())
199                 return STATE_MENU
200             keyboard = build_time_keyboard(filtered_times, busy_slots,
                    normalized_date_str, prefix="time_")
201             if not keyboard.inline_keyboard:
202                 await update.message.reply_text("All slots are occupied on
                        this date. Appointment cancelled.")
203                 await update.message.reply_text("Returning you to the menu."
                        , reply_markup=main_menu_keyboard())
204                 return STATE_MENU
```

```
205             await update.message.reply_text("Select a convenient time:",
                    reply_markup=keyboard)
206             return STATE_SIGNUP_TIME
207             async def sign_up_time_callback(update: Update, context:
                    ContextTypes.DEFAULT_TYPE):
208             """After selecting time, encrypt data, save to DB, and schedule
                    reminder."""
209             query = update.callback_query
210             await query.answer()
211             user_id = query.from_user.id
212             time_str = query.data.split("_", 1)[1]
213             name = context.user_data['name']
214             phone = context.user_data['phone']
215             date_str = context.user_data['date']
216
217             encrypted_name = aes_encrypt(name)
218             encrypted_phone = aes_encrypt(phone)
219
220             appointments[user_id] = {
221                 "name": encrypted_name,
222                 "phone": encrypted_phone,
223                 "date": date_str,
224                 "time": time_str,
225                 "job_id_10min": None,
226                 "job_id_5min": None,
227                 "has_answered_reminder": False
228             }
229             busy_slots[(date_str, time_str)] = user_id
230             name_parts = name.split()
231             last_name = name_parts[0]
232             first_name = name_parts[1]
233             middle_name = " ".join(name_parts[2:]) if len(name_parts) > 2
                    else ""
234
235             try:
236                 client_id = insert_client(last_name, first_name, middle_name
                        , encrypted_phone)
237                 db_appointment_id = insert_appointment(client_id, date_str,
                        time_str, encrypted_name)
238                 appointments[user_id]["db_appointment_id"] =
                        db_appointment_id
239                 insert_status(db_appointment_id, "pending", encrypted_phone,
                         encrypted_name)
240             except Exception as e:
241                 insert_log(f"Error in sign_up_time_callback: {e}")
242
243             moscow_tz = timezone("Europe/Moscow")
244             try:
245                 appt_dt_naive = datetime.strptime(f"{date_str} {time_str}",
                        "%Y-%m-%d %H:%M")
246                 appt_datetime = moscow_tz.localize(appt_dt_naive)
247                 now = datetime.now(moscow_tz)
248                 delta = (appt_datetime - now).total_seconds()
249                 logger.info(f"Time until appointment: {delta} seconds")
250                 if delta >= 10 * 60:
251                     reminder_time = appt_datetime - timedelta(minutes=10)
252                     logger.info(f"Scheduling reminder for user {user_id} at
                            {reminder_time}")
253                     job_10min = context.job_queue.run_once(
254                         send_10min_reminder,
255                         when=reminder_time,
```

```
                        chat_id=user_id,
                        name=f"reminder_{user_id}"
                    )
                    appointments[user_id]["job_id_10min"] = job_10min.job.id
                else:
                    logger.info("Not scheduling reminder: less than 10 minutes remain")
                logger.info(f"Appointment set for: {appt_datetime}")
            except ValueError as ve:
                logger.error(f"Error parsing appointment time: {ve}")
            await query.message.reply_text(
                f"Appointment confirmed!\nName: {name}\nPhone: {phone}\nDate: {date_str}\nTime: {time_str}"
            )
            context.user_data.clear()
            await query.message.reply_text("Returning you to the menu.", reply_markup=main_menu_keyboard())
            return STATE_MENU
            async def send_10min_reminder(context: ContextTypes.DEFAULT_TYPE):
            job = context.job
            user_id = job.chat_id
            logger.info(f"send_10min_reminder triggered for user {user_id} at {datetime.now()}")
            if user_id not in appointments:
                logger.warning(f"send_10min_reminder: user {user_id} not found")
                return
            if appointments[user_id].get("has_answered_reminder"):
                logger.info(f"User {user_id} already answered reminder; skipping")
                return
            msg = await context.bot.send_message(
                chat_id=user_id,
                text=(
                    "Your appointment is starting soon, will you arrive at the scheduled time?\n"
                    "Choose an option:"
                ),
                reply_markup=REMINDER_OPTIONS_KEYBOARD
            )
            logger.info(f"Reminder message sent to user {user_id}")
            job_5min = context.job_queue.run_once(
                resend_reminder,
                when=timedelta(minutes=5),
                chat_id=user_id,
                name=f"reask_{user_id}",
                data={"reminder_message_id": msg.message_id},
            )
            appointments[user_id]["job_id_5min"] = job_5min.job.id

        async def resend_reminder(context: ContextTypes.DEFAULT_TYPE):
            job = context.job
            user_id = job.chat_id
            logger.info(f"resend_reminder triggered for user {user_id} at {datetime.now()}")
            if user_id not in appointments:
                logger.warning(f"resend_reminder: user {user_id} not found")
                return
            if appointments[user_id].get("has_answered_reminder"):
                logger.info(f"User {user_id} answered; not resending")
```

```
            return
        await context.bot.send_message(
            chat_id=user_id,
            text=(
                "You haven't responded yet!\n"
                "Please respond, otherwise, our manager will need to call you for clarification.\n"
                "Think of our manager!\n\nWill you arrive at the scheduled time?"
            ),
            reply_markup=REMINDER_OPTIONS_KEYBOARD
        )
        logger.info(f"Resend reminder message sent to user {user_id}")
        async def reminder_answer_callback(update: Update, context: ContextTypes.DEFAULT_TYPE):
        query = update.callback_query
        user_id = query.from_user.id
        data = query.data
        if user_id not in appointments:
            await query.answer("You have no appointment.")
            return
        appointments[user_id]["has_answered_reminder"] = True
        await query.answer()
        status = None
        if data == "reminder_yes":
            await query.message.reply_text("Great! We're expecting you.")
            status = "pending"
        elif data == "reminder_late":
            await query.message.reply_text("Being late isn't good... but alright. Please arrive within 15 minutes or the appointment will be cancelled!")
            status = "pending"
        elif data == "reminder_no":
            keyboard = [
                [InlineKeyboardButton("Specify reason", callback_data="no_show_reason")],
                [InlineKeyboardButton("Exit", callback_data="no_show_exit")]
            ]
            markup = InlineKeyboardMarkup(keyboard)
            await query.message.reply_text("That's unfortunate! Please specify why you can't come:", reply_markup=markup)
            status = "cancelled"
        elif data == "reminder_here":
            await query.message.reply_text("Wow! You're as fast as a cheetah!")
            status = "finished"
        elif data == "reminder_exit":
            await query.message.reply_text("Returning you to the menu.", reply_markup=main_menu_keyboard())
            return STATE_MENU
        if status and "db_appointment_id" in appointments[user_id]:
            try:
                update_status(appointments[user_id]["db_appointment_id"], status, appointments[user_id]["phone"], appointments[user_id]["name"])
                logger.info(f"Status updated for user {user_id} to {status}")
            except Exception as e:
```

```
                insert_log(f"Error updating status in reminder_answer_callback: {e}")
        return STATE_MENU
        async def no_show_reason_callback(update: Update, context: ContextTypes.DEFAULT_TYPE):
        query = update.callback_query
        await query.answer()
        await query.message.reply_text("Please type your reason for not showing up:")
        return STATE_NO_SHOW_REASON

    async def no_show_exit_callback(update: Update, context: ContextTypes.DEFAULT_TYPE):
        query = update.callback_query
        await query.answer()
        await query.message.reply_text("Returning you to the menu.", reply_markup=main_menu_keyboard())
        return STATE_MENU

    async def no_show_reason_text(update: Update, context: ContextTypes.DEFAULT_TYPE):
        reason = update.message.text.strip()
        user_id = update.effective_user.id
        logger.info(f"No-show reason from {user_id}: {reason}")
        await update.message.reply_text("Reason received, thank you for your feedback.")
        await update.message.reply_text("Returning you to the menu.", reply_markup=main_menu_keyboard())
        return STATE_MENU
        async def cancel_confirm(update: Update, context: ContextTypes.DEFAULT_TYPE):
        user_id = update.effective_user.id
        answer = update.message.text.strip().lower()
        if answer == "yes":
            if user_id in appointments:
                date_str = appointments[user_id]['date']
                time_str = appointments[user_id]['time']
                job_id_10min = appointments[user_id].get("job_id_10min")
                job_id_5min = appointments[user_id].get("job_id_5min")
                if job_id_10min:
                    for j in context.job_queue.get_jobs_by_name(f"reminder_{user_id}"):
                        j.schedule_removal()
                if job_id_5min:
                    for j in context.job_queue.get_jobs_by_name(f"reask_{user_id}"):
                        j.schedule_removal()
                if (date_str, time_str) in busy_slots:
                    del busy_slots[(date_str, time_str)]
                del appointments[user_id]
                await update.message.reply_text("Appointment cancelled.")
            else:
                await update.message.reply_text("You have no appointment.")
        else:
            await update.message.reply_text("Appointment cancellation not confirmed.")
        await update.message.reply_text("Returning you to the menu.", reply_markup=main_menu_keyboard())
        return STATE_MENU
```

```
400             async def change_name(update: Update, context: ContextTypes.
                    DEFAULT_TYPE):
401             user_id = update.effective_user.id
402             new_name = update.message.text.strip()
403             if new_name != "-":
404                 parts = new_name.split()
405                 if len(parts) < 2:
406                     await update.message.reply_text("The name must contain
                            at least 2 words. Changes cancelled.")
407                     await update.message.reply_text("Returning you to the
                            menu.", reply_markup=main_menu_keyboard())
408                     return STATE_MENU
409                 appointments[user_id]['name'] = aes_encrypt(new_name)
410             await update.message.reply_text("Enter new date (or '-' to skip)
                    :")
411             return STATE_CHANGE_DATE
412             async def change_date(update: Update, context: ContextTypes.
                    DEFAULT_TYPE):
413             user_id = update.effective_user.id
414             new_date = update.message.text.strip()
415             old_date = appointments[user_id]['date']
416             old_time = appointments[user_id]['time']
417             if new_date != "-":
418                 parsed_date = None
419                 for fmt in ["%Y-%m-%d", "%d.%m.%Y"]:
420                     try:
421                         parsed_date = datetime.strptime(new_date, fmt)
422                         break
423                     except ValueError:
424                         pass
425                 if not parsed_date:
426                     await update.message.reply_text("Unrecognized date
                            format. Changes cancelled.")
427                     await update.message.reply_text("Returning you to the
                            menu.", reply_markup=main_menu_keyboard())
428                     return STATE_MENU
429                 today = datetime.now().date()
430                 if parsed_date.date() < today:
431                     await update.message.reply_text("Oops! This date has
                            already passed. Changes cancelled.")
432                     await update.message.reply_text("Returning you to the
                            menu.", reply_markup=main_menu_keyboard())
433                     return STATE_MENU
434                 if (old_date, old_time) in busy_slots and busy_slots[(
                        old_date, old_time)] == user_id:
435                     del busy_slots[(old_date, old_time)]
436                 new_date_str = parsed_date.strftime("%Y-%m-%d")
437                 appointments[user_id]['date'] = new_date_str
438             else:
439                 new_date_str = old_date
440             job_id_10min = appointments[user_id].get("job_id_10min")
441             job_id_5min = appointments[user_id].get("job_id_5min")
442             if job_id_10min:
443                 for j in context.job_queue.get_jobs_by_name(f"reminder_{
                        user_id}"):
444                     j.schedule_removal()
445                 appointments[user_id]["job_id_10min"] = None
446             if job_id_5min:
447                 for j in context.job_queue.get_jobs_by_name(f"reask_{user_id
                        }"):
448                     j.schedule_removal()
```

```
449                 appointments[user_id]["job_id_5min"] = None
450             appointments[user_id]["has_answered_reminder"] = False
451             parsed_new_date = datetime.strptime(new_date_str, "%Y-%m-%d")
452             today_dt = datetime.now().date()
453             now = datetime.now()
454             if parsed_new_date.date() == today_dt:
455                 current_time = now.time()
456                 filtered_times = [t for t in AVAILABLE_TIMES if time(*map(
                        int, t.split(":"))) > current_time]
457             else:
458                 filtered_times = AVAILABLE_TIMES
459             keyboard = build_time_keyboard(filtered_times, busy_slots,
                    new_date_str, prefix="change_time_")
460             if not keyboard.inline_keyboard:
461                 await update.message.reply_text("All slots are occupied or
                        the time has passed. Changes cancelled.")
462                 await update.message.reply_text("Returning you to the menu."
                        , reply_markup=main_menu_keyboard())
463                 return STATE_MENU
464             await update.message.reply_text("Select new time:", reply_markup
                    =keyboard)
465             return STATE_CHANGE_TIME
466             async def change_time_callback(update: Update, context:
                    ContextTypes.DEFAULT_TYPE):
467             query = update.callback_query
468             await query.answer()
469             user_id = query.from_user.id
470             new_time = query.data.split("_", 2)[2]
471             new_date = appointments[user_id]['date']
472             appointments[user_id]['time'] = new_time
473             busy_slots[(new_date, new_time)] = user_id
474             try:
475                 moscow_tz = timezone("Europe/Moscow")
476                 appt_dt_naive = datetime.strptime(f"{new_date} {new_time}",
                        "%Y-%m-%d %H:%M")
477                 appt_datetime = moscow_tz.localize(appt_dt_naive)
478                 now = datetime.now(moscow_tz)
479                 delta = (appt_datetime - now).total_seconds()
480                 if delta >= 10 * 60:
481                     job_10min = context.job_queue.run_once(
482                         send_10min_reminder,
483                         when=appt_datetime - timedelta(minutes=10),
484                         chat_id=user_id,
485                         name=f"reminder_{user_id}"
486                     )
487                     appointments[user_id]["job_id_10min"] = job_10min.job.id
488                 else:
489                     appointments[user_id]["job_id_10min"] = None
490             except ValueError:
491                 pass
492             if "db_appointment_id" in appointments[user_id]:
493                 try:
494                     update_appointment(
495                         appointments[user_id]["db_appointment_id"],
496                         new_date,
497                         new_time,
498                         appointments[user_id]["name"]
499                     )
500                     logger.info(f"Appointment updated in DB for user {
                            user_id}")
501                 except Exception as e:
```

```
502                     insert_log(f"Error updating appointment in
                            change_time_callback: {e}")
503             await query.message.reply_text(
504                 f"Appointment updated:\nName: {aes_decrypt(appointments[
                        user_id]['name'])}\nPhone: {aes_decrypt(appointments[
                        user_id]['phone'])}\nDate: {appointments[user_id]['date
                        ']}\nTime: {appointments[user_id]['time']}"
505             )
506             await query.message.reply_text("Returning you to the menu.",
                    reply_markup=main_menu_keyboard())
507             return STATE_MENU
508         async def cancel_conversation(update: Update, context: ContextTypes.
                DEFAULT_TYPE):
509             await update.message.reply_text("Action cancelled. Returning you
                    to the menu.", reply_markup=main_menu_keyboard())
510             return STATE_MENU
511             async def cancel_conversation(update: Update, context:
                    ContextTypes.DEFAULT_TYPE):
512             await update.message.reply_text("Action cancelled. Returning you
                    to the menu.", reply_markup=main_menu_keyboard())
513             return STATE_MENU
514
515         def run_java_app():
516             compile_process = subprocess.run(
517                 [
518                     "javac",
519                     "Java/db_parser/src/main/java/ru/spbstu/telematics/java/
                            App.java"
520                 ],
521                 capture_output=True,
522                 text=True
523             )
524
525             if compile_process.returncode != 0:
526                 print("Compilation error:")
527                 print(compile_process.stderr)
528                 return
529             else:
530                 print("Compilation successful.")
531
532             run_process = subprocess.Popen(
533                 [
534                     "java",
535                     "-cp",
536                     "Java/db_parser/src/main/java",
537                     "ru.spbstu.telematics.java.App"
538                 ],
539                 stdout=subprocess.PIPE,
540                 stderr=subprocess.PIPE,
541                 text=True
542             )
543
544             stdout, stderr = run_process.communicate()
545             if run_process.returncode != 0:
546                 print("Error executing Java application:")
547                 print(stderr)
548             else:
549                 print("Java application output:")
550                 print(stdout)
551
552         async def run_java_parser(context: ContextTypes.DEFAULT_TYPE):
```

```
553             logger.info("Running Java parser to update JSON via Maven")
554             compile_process = subprocess.run(
555                 ["mvn", "compile"],
556                 capture_output=True,
557                 text=True,
558                 cwd="Java/db_parser"
559             )
560             if compile_process.returncode != 0:
561                 logger.error("Maven compilation error:")
562                 logger.error(compile_process.stderr)
563                 return
564
565             run_process = subprocess.run(
566                 ["mvn", "exec:java", "-Dexec.mainClass=ru.spbstu.telematics.
                    java.App"],
567                 capture_output=True,
568                 text=True,
569                 cwd="Java/db_parser"
570             )
571             if run_process.returncode != 0:
572                 logger.error("Error running Java parser via Maven:")
573                 logger.error(run_process.stderr)
574             else:
575                 logger.info("Parser updated JSON:")
576                 logger.info(run_process.stdout)
577                 def main():
578                 from dotenv import load_dotenv
579                 load_dotenv("/Users/ayzek/Desktop/Ayzek/Lyubimka/OPD - 2
                    course/TelegramBot/.env")
580                 logger.info(f"Current time: {datetime.now()}")
581                 TOKEN = os.getenv("TELEGRAM_API_TOKEN")
582                 if not TOKEN:
583                     raise ValueError("TELEGRAM_API_TOKEN not found in .env
                        file")
584                 application = Application.builder().token(TOKEN).build()
585
586                 async def test_job(ctx: ContextTypes.DEFAULT_TYPE):
587                     logger.info("Test job triggered")
588
589                 application.job_queue.run_once(test_job, when=timedelta(
                    seconds=10))
590                 application.job_queue.run_repeating(run_java_parser,
                    interval=120, first=5)
591
592                 conv_handler = ConversationHandler(
593                     entry_points=[CommandHandler("start", info_command)],
594                     states={
595                         STATE_MENU: [CallbackQueryHandler(menu_callback)],
596                         STATE_SIGNUP_NAME: [MessageHandler(filters.TEXT & ~
                            filters.COMMAND, sign_up_name)],
597                         STATE_SIGNUP_PHONE: [MessageHandler(filters.TEXT & ~
                            filters.COMMAND, sign_up_phone)],
598                         STATE_SIGNUP_DATE: [MessageHandler(filters.TEXT & ~
                            filters.COMMAND, sign_up_date)],
599                         STATE_SIGNUP_TIME: [CallbackQueryHandler(
                            sign_up_time_callback, pattern=r"^time_")],
600                         STATE_CANCEL_CONFIRM: [MessageHandler(filters.TEXT &
                             ~filters.COMMAND, cancel_confirm)],
601                         STATE_CHANGE_NAME: [MessageHandler(filters.TEXT & ~
                            filters.COMMAND, change_name)],
```

```
602                         STATE_CHANGE_DATE: [MessageHandler(filters.TEXT & ~
                                filters.COMMAND, change_date)],
603                         STATE_CHANGE_TIME: [CallbackQueryHandler(
                                change_time_callback, pattern=r"^change_time_")],
604                         STATE_NO_SHOW_REASON: [MessageHandler(filters.TEXT &
                                ~filters.COMMAND, no_show_reason_text)],
605                     },
606                     fallbacks=[CommandHandler("cancel", cancel_conversation)
                            ],
607                 )
608 
609                 application.add_handler(CommandHandler("info", start_command
                        ))
610                 application.add_handler(CallbackQueryHandler(
                        reminder_answer_callback, pattern=r"^reminder_"))
611                 application.add_handler(CallbackQueryHandler(
                        no_show_reason_callback, pattern=r"^no_show_reason"))
612                 application.add_handler(CallbackQueryHandler(
                        no_show_exit_callback, pattern=r"^no_show_exit"))
613                 application.add_handler(conv_handler)
614 
615                 application.run_polling()
616 
617             if __name__ == "__main__":
618                 main()
```

3 Схема БД

На рис. 2 представлена схема базы данных. Название базы данных: booking_system.

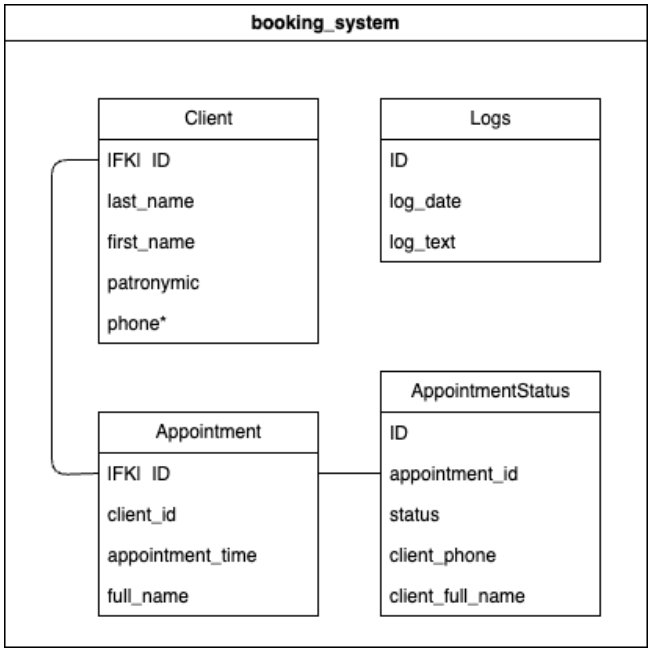

Рис. 2: Схема базы данных.

3.1 Чтение схемы БД

В СУБД MySQL, хранится база данных booking_system. В ней четыре таблицы:

1. Client FK-id
2. Appointment FK-id
3. AppointmentStatus PK-appointment_id
4. Logs PK-id

4 Реализация парсера

4.1 Парсер

Парсер - Программное обеспечение или модуль который считывает текст или другой поток данных и преобразует его в желаемую форму или формат текста. В реализации используется собственный парсер написанный на языке программирования Java.

4.2 Реализация

```
    package ru.spbstu.telematics.java;

import java.io.File;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import com.fasterxml.jackson.databind.ObjectMapper;

public class App {
    private static final String URL = "jdbc:mysql://localhost:3306/
        booking_system?useSSL=false&serverTimezone=UTC";
    private static final String USERNAME = "root";
    private static final String PASSWORD = "Ayzek123321";

    public static void main(String[] args) {
        List<Map<String, Object>> clients = new ArrayList<>();
        List<Map<String, Object>> appointments = new ArrayList<>();
        List<Map<String, Object>> appointmentStatuses = new ArrayList<>();

        Connection connection = null;
        Statement stmt = null;

        try {
            Class.forName("com.mysql.cj.jdbc.Driver");
            connection = DriverManager.getConnection(URL, USERNAME, PASSWORD
                );
            stmt = connection.createStatement();

            ResultSet rs = stmt.executeQuery("SELECT * FROM Client");
            while (rs.next()) {
                Map<String, Object> row = new HashMap<>();
                row.put("id", rs.getInt("id"));
                row.put("last_name", rs.getString("last_name"));
                row.put("first_name", rs.getString("first_name"));
                row.put("patronymic", rs.getString("patronymic"));
                row.put("phone", rs.getString("phone"));
                clients.add(row);
            }
            rs.close();

            rs = stmt.executeQuery("SELECT * FROM Appointment");
            while (rs.next()) {
                Map<String, Object> row = new HashMap<>();
```

```
                row.put("id", rs.getInt("id"));
                row.put("client_id", rs.getInt("client_id"));
                row.put("appointment_date", rs.getDate("appointment_date"));
                row.put("appointment_time", rs.getTime("appointment_time"));
                row.put("full_name", rs.getString("full_name"));
                appointments.add(row);
            }
            rs.close();

            rs = stmt.executeQuery("SELECT * FROM AppointmentStatus");
            while (rs.next()) {
                Map<String, Object> row = new HashMap<>();
                row.put("id", rs.getInt("id"));
                row.put("appointment_id", rs.getInt("appointment_id"));
                row.put("status", rs.getString("status"));
                row.put("client_phone", rs.getString("client_phone"));
                row.put("client_full_name", rs.getString("client_full_name")
                    );
                appointmentStatuses.add(row);
            }
            rs.close();

            Map<String, Object> data = new HashMap<>();
            data.put("clients", clients);
            data.put("appointments", appointments);
            data.put("appointmentStatus", appointmentStatuses);

            ObjectMapper mapper = new ObjectMapper();
            mapper.writerWithDefaultPrettyPrinter().writeValue(new File("
                booking_system.json"), data);

            System.out.println("JSON file created successfully:
                booking_system.json");

        } catch (Exception e) {
            e.printStackTrace();
        } finally {

            if (stmt != null) {
                try {
                    stmt.close();
                } catch (SQLException se2) {
                    se2.printStackTrace();
                }
            }
            if (connection != null) {
                try {
                    connection.close();
                } catch (SQLException se) {
                    se.printStackTrace();
                }
            }
        }
    }
}
```

5 Схема Maven проекта

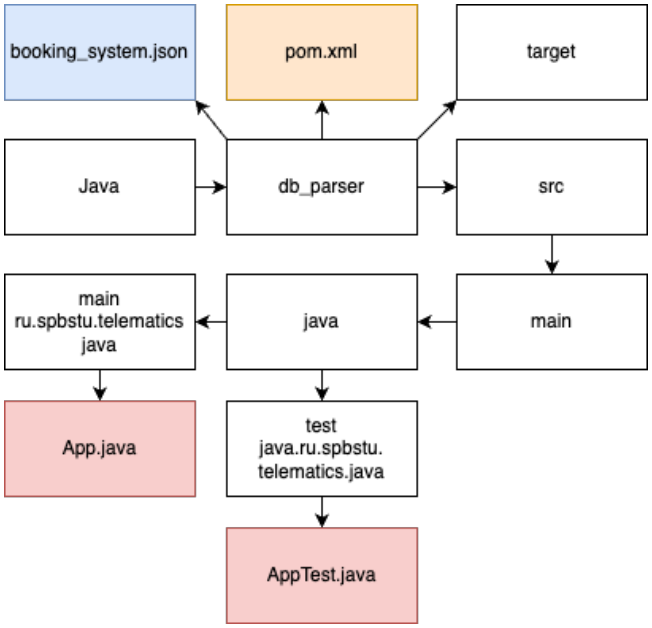

Рис. 3: Схема проекта Maven.

6 Реализация Maven проекта

Для реализации проекта и его запуска нужны две команды. Так как в pom.xml, главный класс уже прописан как класс парсера.

```
    mvn build
    mvn compile
    mvn exec:java
```

6.1 pom.xml

pom.xml - файл моделирования и структурирования проекта. В реализации использовалась библиотека sql, fasterxml.jackson.databind.ObjectMapper

```
    <project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>ru.spbstu.telematics.java</groupId>
  <artifactId>db_parser</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>db_parser</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>com.fasterxml.jackson.core</groupId>
      <artifactId>jackson-databind</artifactId>
      <version>2.15.2</version>
    </dependency>
    <dependency>
      <groupId>com.fasterxml.jackson.core</groupId>
      <artifactId>jackson-core</artifactId>
      <version>2.15.2</version>
    </dependency>
    <dependency>
      <groupId>com.fasterxml.jackson.core</groupId>
      <artifactId>jackson-annotations</artifactId>
      <version>2.15.2</version>
    </dependency>
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>8.0.33</version>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>exec-maven-plugin</artifactId>
        <version>3.1.0</version>
        <configuration>
```

```
            <mainClass>ru.spbstu.telematics.java.App</mainClass>
          </configuration>
        </plugin>
      </plugins>
    </build>
</project>
```

7 Реализация Conda проекта

Для реализации Conda проекта, была прописана команда в директории файлов:

```
    conda build
```

7.1 setup.py

Файл setup.py, служит конфигурацией для проекта Conda:

```
    from setuptools import setup, find_packages
setup(
    name="tg_bot_for_clients_lb",
    version="0.1.0",
    packages=find_packages(),
    install_requires=[
        "python-telegram-bot",
        "mysql-connector-python",
        "python-dotenv",
        "pycryptodome",
    ],
)
```

Заключение

В заключении, был создан Conda проект телеграм-бота. Была создана база данных в СУБД MySQL. Запросы на записи данных шифрованы. При получении данных они дешифруются. Был реализован проект Maven, в котором реалзиован парсер, считывающий данные из базы данных, и создающий файл с дешифрованными данными.

Конфигурация ПО для выдачи:

- Beta-версия.
- v1.0.1.

Контактная информация

- Телефон: +7 (921) 945-67-03
- Электронная почта: thisisnauchno@gmail.com
- Корпоративная почта: ayzek@thebloomsbridge.com
- Telegram: @undefined_1010
- WhatsApp: +7 (921) 945-67-03
- Сайт компании, предоставляющей услуги ПО: thebloomsbridge.io
- Сайт с репозиториями автора: github.com/MathematicLove

Салимли Айзек Мухтар оглы The Blooms Bridge Software, Machine Learning Engineer.