Документация по итоговому проекту по дисциплине “Управление IT-проектами”

В проекте на тему “Телеграм-бот - помощник в составлении резюме” участвовали студенты Б9122-02.03.01сцт:

1) **Бекболот Отгонцэцэг**
   Менеджер
   Управляла ходом работы, направляла команду на правильные идеи и мысли, оформляла заметки для презентации уже готового продукта, помечала, что нужно выполнить до конечного выполнения задания в Figma.
2) **Ильяхова Алиса**
   Дизайнер
   Продумала и выполнила дизайн логотипа, выбрала фирменные цвета и шрифт в Figma, Adobe illustrator 2022.
3) **Протопопова Анастасия**
   Разработчик
   Выполнила написания кода на PyCharm, работала с Telegram.

Введение:

Правильно составленное резюме - ключ к успешному трудоустройству. Многие соискатели испытывают трудности в его создании: как выделить навыки, оформить документ и указать достижения. Наш Телеграм-бот поможет упростить этот процесс, предлагая пошаговые инструкции, шаблоны и полезные советы для создания эффективного резюме.

Цель:

Основная цель проекта - разработать Телеграм-бота, который поможет пользователям создавать профессиональные резюме. Бот будет предоставлять пошаговые инструкции, шаблоны и советы, чтобы сделать процесс составления резюме более доступным и эффективным для соискателей разных уровней.

Интересный факт:

Согласно исследованиям, более 60% работодателей принимают решение о том, приглашать ли кандидата на собеседование, основываясь только на резюме. Это подчеркивает важность качественного и правильно составленного документа для успешного трудоустройства.

Главная “боль” аудитории:

Неуверенность в своих навыках и недостаток знаний о том, как правильно представить свой опыт и достижения, создают стресс и затруднения при поиске работы.

Проблема:

Соискатели сталкиваются с трудностями в создании эффективного резюме, что приводит к низким шансам на успешное трудоустройство.

Решение:

Разработать Телеграм-бот с интеграцией интеллектуальной модели, такой как GigaChat, помогающий пользователям не только составить резюме, но и улучшить его качество, адаптируя текст под требования конкретной вакансии и предлагая рекомендации по улучшению.

Аргументация выгод и преимуществ предложения:

- Удобство и доступность:
  Пошаговое руководство: Бот предлагает простые и понятные инструкции, что делает процесс создания резюме доступным даже для тех, кто не имеет опыта в этом.
  Доступ в любое время: Пользователи могут обращаться к боту в любое время и из любого места, что удобно для занятых соискателей.
- Экономия времени:
  Автоматизация процесса: Бот поможет быстро собрать всю необходимую информацию и оформить резюме без необходимости искать шаблоны и советы в интернете.
  Готовые шаблоны: Пользователи могут выбрать из множества готовых шаблонов, что существенно сокращает время на оформление.
- Качество и профессионализм:
  Советы по оптимизации: Бот предоставляет рекомендации по улучшению резюме, что повышает его качество и привлекательность для работодателей.
  Адаптация под вакансии: Возможность адаптировать резюме под конкретные вакансии увеличивает шансы на успешное трудоустройство.

4 почему:

1) Почему мы хотим разработать телеграм-бот по созданию резюме?
   Потому что это позволит нам предложить пользователям удобный и современный инструмент для создания резюме.
2) Почему мы думаем, что это будет удобно для пользователей?
   Потому что многие люди предпочитают использовать Telegram для общения и взаимодействия с различными сервисами.
3) Почему мы думаем, что это будет современным инструментом?
   Потому что технология чат-ботов активно развивается и становится все более популярной.
4) Почему мы думаем, что это будет полезно для пользователей?
   Потому что создание резюме зачастую занимает много времени и требует специальных навыков, а бот сможет автоматизировать этот процесс.

4 силы:

1) Технологическое развитие: Быстрый рост и развитие технологий чат-ботов, что делает возможным создание сложных и функциональных решений.
2) Популярность Telegram: Большая популярность мессенджера Telegram в России, что обеспечивает высокую вероятность использования бота пользователями.
3) Современные инструменты разработки: Существование множества инструментов и библиотек для создания ботов, что облегчает процесс разработки и тестирования.
4) Ресурсы и компетенции: Наличие специалистов и разработчиков, обладающих навыками и опытом в создании подобных проектов, что гарантирует высокое качество конечного продукта.

SMART:

Specific:

1) Создать телеграмм-бота для помощи пользователям в составлении профессионального резюме.
2) Обеспечить простоту использования и интуитивность интерфейса бота.
3) Внедрить функции автоматического заполнения данных и генерацию резюме в формате PDF.

Measurable:

1) Количество пользователей, которые воспользовались ботом в течение месяца.
2) Процент ошибок и отказов в работе бота.
3) Время, затраченное пользователем на создание резюме с помощью бота.

Achievable:

1) Провести тестирование и отладку бота перед запуском.

Relevant:

1) Соответствие требованиям пользователей по созданию резюме.
2) Удовлетворение потребностей работодателей в найме квалифицированных кандидатов.
3) Повышение конкурентоспособности на рынке труда.

Time bound:

1) Запустить бета-версию бота.
2) Завершить полную разработку и запуск бота в течение трех месяцев.
3) Регулярно обновлять и улучшать функционал бота каждые три месяца.

SWOT:

Strengths:

1) Автоматизация процесса создания резюме.
2) Простота использования и интуитивный интерфейс.

3) Гибкость и возможность персонализации резюме.

Weaknesses:

1) Ограниченная известность по сравнению с крупными игроками.
2) Отсутствие интеграции с популярными ресурсами по трудоустройству.
3) Необходимость продвижения и маркетинговой активности.

Opportunities:

1) Увеличение популярности Telegram как платформы для бизнес-приложений.
2) Интеграция с локальными ресурсами по трудоустройству.
3) Возможность расширения функционала за счет интеграции с Al-технологиями

Threats:

1) Высокая конкуренция на рынке.
2) Изменения в политике Telegram, влияющие на работу ботов.
3) Возможное появление новых конкурентов с аналогичным функционалом.

Логотип:

Выполнения кода:

1) Импорт библиотек:
   - logging: Для ведения журналов (логирования).
   - os: Для работы с операционной системой (например, для получения переменных окружения).
   - aiogram: Библиотека для создания ботов в Telegram.
   - dotenv: Для загрузки переменных окружения из .env файла.

- langchain: Для работы с языковыми моделями (в данном случае GigaChat).
- reportlab: Для создания PDF-документов.
- requests и BeautifulSoup: Для работы с HTTP-запросами и парсинга HTML (в данном коде не используются, но могут быть полезны в будущем).

2) Настройка:
   - promptone: Строка, описывающая цель бота - формирование резюме.
   - load_dotenv(): Загружает переменные окружения из файла .env, что позволяет хранить чувствительные данные (например, токены) вне кода.
   - Настройка логирования на уровень INFO для отслеживания событий в приложении.
   - Получение токена Telegram и ключа авторизации для GigaChat из переменных окружения.
3) Инициализация:
   - Создание экземпляра бота с использованием токена.
   - Настройка хранилища состояний (в данном случае используется память).
   - Создание диспетчера для обработки сообщений и состояний.
   - Добавление middleware для логирования событий.
   - Инициализация GigaChat с использованием ключа авторизации.
4) ask_gigachat:
   - Асинхронная функция для отправки сообщений в GigaChat и получения ответа. Она использует метод invoke объекта chat.
5) wrap_text:
   - Функция для разбивки текста на строки с учетом максимальной длины строки. Это нужно для корректного отображения текста в PDF-документе.
6) create_pdf:
   - Функция для создания PDF-документа с заголовком “Резюме”. Она использует ReportLab для оформления текста и добавления его в документ.
   - Регистрация шрифта SegoeUI и установка начального шрифта для документа.
7) await state.update_data(skills=message.text):
   - Эта строка обновляет состояние пользователя, добавляя информацию о навыках (skills). Значение message.text - это текст сообщения, которое пользователь отправил боту. Таким образом, вы сохраняете введенные пользователем навыки в хранилище состояний.

   await message.answer("Укажите ваши пожелания к работе."):

   - Бот отправляет сообщение пользователю с просьбой указать его пожелания к работе. Это текстовое сообщение будет отображено в чате.

   await ResumeForm.wishes.set():

- Здесь вы устанавливаете новое состояние для конечного автомата (FSM). В данном случае вы переходите к состоянию, где пользователь должен ввести свои пожелания к работе.

8) @dp.message_handler(state=ResumeForm.wishes): Декоратор определяет, что эта функция будет вызываться, когда бот находится в состоянии wishes.
   await state.update_data(wishes=message.text): Сохраняет введенные пользователем пожелания в состоянии.
   await message.answer("Укажите желаемую зарплату."): Отправляет пользователю сообщение с просьбой указать желаемую зарплату.
   await ResumeForm.salary.set(): Переходит к следующему состоянию, ожидая ввод зарплаты.
9) @dp.message_handler(state=ResumeForm.salary): Этот обработчик срабатывает, когда бот ожидает ввод зарплаты.
   await state.update_data(salary=message.text): Сохраняет введенную зарплату.
   user_data = await state.get_data(): Получает все данные, собранные до этого момента.
   Формирование сообщения: Создается список сообщений, который включает все собранные данные о пользователе и запрос на генерацию резюме.
   generated_resume = await ask_gigachat(messages): Отправляет собранные данные на генерацию резюме с помощью внешнего API или функции.
   Генерация PDF: Создается PDF-файл с резюме и отправляется пользователю вместе с текстом резюме.
   await ResumeForm.finalize.set(): Устанавливает состояние для финализации процесса.
10) Декоратор: Этот обработчик срабатывает, когда пользователь отправляет сообщение "Редактировать" и находится в состоянии finalize.
   Сообщение: Бот отвечает пользователю с вопросом о том, какое поле он хочет отредактировать, и прикрепляет клавиатуру, созданную функцией get_edit_menu().
   Установка состояния: Устанавливается состояние edit, что позволяет боту ожидать дальнейшего ввода от пользователя.
11) Состояние: Этот обработчик срабатывает, когда бот находится в состоянии edit.
   Проверка поля: Получает текст сообщения от пользователя и проверяет, является ли он допустимым полем для редактирования.
   Обновление состояния: Если поле допустимо, оно сохраняется в состоянии, и бот запрашивает новое значение. Если пользователь выбрал "отмена", процесс редактирования прекращается. Если введено неверное поле, бот сообщает об этом.
12) Формирование сообщений: Создается список сообщений для отправки системе (или внешнему API), где содержится как инструкция (от системы), так и обновленные данные пользователя (от человека).

Использование значений: При формировании сообщения используется метод get для безопасного извлечения значений из словаря с указанием значения по умолчанию ("не указано").

13) Имя файла: Создается строка filename, которая формируется на основе имени пользователя (из данных updated_user_data) и добавляется суффикс _resume.pdf. Это будет имя файла для сохраненного резюме.
Создание PDF: Вызывается функция create_pdf, которая принимает сгенерированное резюме (generated_resume) и имя файла (pdf_filename) для сохранения PDF-документа.

14) Сбор отзыва: Этот обработчик активируется в состоянии feedback и получает текст отзыва от пользователя.
Логирование: Отзыв логируется с помощью модуля logging.
Ответ пользователю: Бот благодарит пользователя за отзыв и сообщает о возможности задать дополнительные вопросы.
Завершение состояния: Состояние завершается с помощью метода finish().

15) Запуск: Этот блок кода проверяет, является ли скрипт основным модулем, и запускает бота с помощью метода start_polling(), который начинает опрашивать обновления от Telegram.

Листинг программы:

```
import logging
import os
from aiogram import Bot, Dispatcher, types
from aiogram.contrib.middlewares.logging import LoggingMiddleware
from aiogram.dispatcher import FSMContext
from aiogram.dispatcher.filters.state import State, StatesGroup
from aiogram.contrib.fsm_storage.memory import MemoryStorage
from aiogram.utils import executor
from dotenv import load_dotenv
from langchain.schema import HumanMessage, SystemMessage
from langchain_community.chat_models.gigachat import GigaChat
from aiogram.types import ReplyKeyboardMarkup, KeyboardButton
from reportlab.lib.pagesizes import letter
from reportlab.pdfgen import canvas
import re
from reportlab.pdfbase import pdfmetrics
from reportlab.pdfbase.ttfonts import TTFont
import requests
from bs4 import BeautifulSoup

promptone = "Вы — бот, формирующий резюме на основе предоставленных данных. Используйте только полученную информацию, избегая откровенной лжи. Стремитесь к ясности и убедительности изложения, допускайте легкое

# Загружаем переменные окружения из .env файла
load_dotenv()

# Логгирование
logging.basicConfig(level=logging.INFO)

# Токен и данные для GigaChat API
TELEGRAM_TOKEN = os.getenv("TELEGRAM_TOKEN")
GIGACHAT_AUTH_KEY = os.getenv("GIGACHAT_AUTH_KEY")

# Инициализация бота, диспетчера и хранилища состояний
bot = Bot(token=TELEGRAM_TOKEN)
storage = MemoryStorage()
dp = Dispatcher(bot, storage=storage)
dp.middleware.setup(LoggingMiddleware())

# Инициализация GigaChat
chat = GigaChat(credentials=GIGACHAT_AUTH_KEY, verify_ssl_certs=False)


# Определение состояний для FSM
class ResumeForm(StatesGroup):
    name = State()
    phone = State()
    email = State()
    status = State()
    experience = State()
    skills = State()
    wishes = State()
    salary = State()
    finalize = State()
    edit = State()
    evaluation = State()
    waiting_for_new_value = State()
    feedback = State()


# Функция для общения с GigaChat
async def ask_gigachat(messages):
    res = chat.invoke(messages)
    return res.content
```

```
# Функция для генерации PDF с ReportLab
def wrap_text(text, max_length):
    words = text.split(' ')
    lines = []
    current_line = ""

    for word in words:
        # Проверяем, не превышает ли добавление нового слова максимально допустимую длину строки
        if len(current_line) + len(word) + 1 > max_length:
            # Если строка уже не пустая, добавляем её в список линий
            if current_line:
                lines.append(current_line)
            current_line = word  # Начинаем новую строку с текущего слова
        else:
            if current_line:  # Если это не первая добавляемая строка
                current_line += ' ' + word
            else:
                current_line = word  # Первая строка

    # Добавляем оставшуюся строку, если она не пустая
    if current_line:
        lines.append(current_line)

    return lines

# Функция для генерации PDF в деловом стиле
def create_pdf(resume_text, file_name):
    pdfmetrics.registerFont(TTFont('SegoeUI', 'SegoeUI.ttf'))

    # Создаем PDF-документ с использованием canvas
    pdf_file = canvas.Canvas(file_name, pagesize=letter)
    pdf_file.setFont("SegoeUI", 12)

    # Заголовок резюме
    pdf_file.setFont("SegoeUI", 18)
    pdf_file.drawString(100, 750, "Резюме")
    pdf_file.setFont("SegoeUI", 12)
    pdf_file.drawString(100, 730, "==================")

    # Разделяем текст на строки, учитывая форматирование
    lines = resume_text.split('\n')
    y_position = 700

    for line in lines:
        # Обработка символов форматирования
        line = line.replace("**", "")  # Удаляем символы жирного шрифта
        line = line.replace("*", "")  # Удаляем символы курсива
        line = line.replace("```", "")  # Удаляем символы блочного кода
        line = line.replace("`", "")  # Удаляем символы инлайн-кода

        # Разбиваем длинные предложения
        wrapped_lines = wrap_text(line, max_length=75)  # Установите максимальную длину строки

        for wrapped_line in wrapped_lines:
            pdf_file.drawString(100, y_position, wrapped_line)
            y_position -= 15  # Отступ между строками

        # Добавляем дополнительный отступ после каждой секции
        y_position -= 7

    # Заканчиваем генерацию PDF
    pdf_file.save()
```

```
# Функция отправки приветственного сообщения и начала сбора данных
async def send_welcome(message: types.Message):
    await message.answer(
        "Здравствуйте! Я помогу вам составить резюме. Пожалуйста, воспользуйтесь кнопками для навигации.")
    await message.answer("Нажмите 'Начать', чтобы начать процесс создания резюме.", reply_markup=get_main_menu())


# Хендлер для команды /start
@dp.message_handler(commands=['start'])
async def start(message: types.Message):
    await send_welcome(message)


# Создание кнопок
def get_main_menu():
    markup = ReplyKeyboardMarkup(resize_keyboard=True)
    start_button = KeyboardButton("Начать")
    edit_button = KeyboardButton("Редактировать")
    evaluate_button = KeyboardButton("Оценить резюме")
    vacancy_button = KeyboardButton("Вакансия")  # Новая кнопка
    markup.add(start_button, edit_button, evaluate_button, vacancy_button)
    return markup


# Хендлер для обработки нажатия кнопок
@dp.message_handler(lambda message: message.text == "Начать")
async def initiate_resume(message: types.Message):
    await message.answer("Как вас зовут?")
    await ResumeForm.name.set()


# Хендлеры для сбора информации о резюме
@dp.message_handler(state=ResumeForm.name)
async def collect_name(message: types.Message, state: FSMContext):
    await state.update_data(name=message.text)
    await message.answer("Укажите ваш номер телефона.")
    await ResumeForm.phone.set()


# Функция валидации номера телефона
def validate_phone(phone):
    phone_pattern = r"^\+?\d{10,13}$"
    return bool(re.match(phone_pattern, phone))


@dp.message_handler(state=ResumeForm.phone)
async def collect_phone(message: types.Message, state: FSMContext):
    phone = message.text
    if validate_phone(phone):
        await state.update_data(phone=phone)
        await message.answer("Введите ваш адрес электронной почты.")
        await ResumeForm.email.set()
    else:
        await message.answer("Неверный формат номера телефона. Пожалуйста, введите номер в формате +7XXXXXXXXXX.")


# Функция валидации email
def validate_email(email):
    email_pattern = r"^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$"
    return bool(re.match(email_pattern, email))
```

```
@dp.message_handler(state=ResumeForm.email)
async def collect_email(message: types.Message, state: FSMContext):
    email = message.text
    if validate_email(email):
        await state.update_data(email=email)
        await message.answer("Какой у вас статус? (например, 'В активном поиске работы', 'Рассмотрю предложения')")
        await ResumeForm.status.set()
    else:
        await message.answer("Неверный формат электронной почты. Пожалуйста, введите корректный адрес.")


@dp.message_handler(state=ResumeForm.status)
async def collect_status(message: types.Message, state: FSMContext):
    await state.update_data(status=message.text)
    await message.answer("Расскажите о своем опыте работы.")
    await ResumeForm.experience.set()


@dp.message_handler(state=ResumeForm.experience)
async def collect_experience(message: types.Message, state: FSMContext):
    await state.update_data(experience=message.text)
    await message.answer("Теперь укажите свои навыки.")
    await ResumeForm.skills.set()


@dp.message_handler(state=ResumeForm.skills)
async def collect_skills(message: types.Message, state: FSMContext):
    await state.update_data(skills=message.text)
    await message.answer("Укажите ваши пожелания к работе.")
    await ResumeForm.wishes.set()


@dp.message_handler(state=ResumeForm.wishes)
async def collect_wishes(message: types.Message, state: FSMContext):
    await state.update_data(wishes=message.text)
    await message.answer("Укажите желаемую зарплату.")
    await ResumeForm.salary.set()


@dp.message_handler(state=ResumeForm.salary)
async def collect_salary(message: types.Message, state: FSMContext):
    await state.update_data(salary=message.text)

    user_data = await state.get_data()

    # Подготовка данных для генерации резюме
    messages = [
        SystemMessage(content=promptone),
        HumanMessage(content=f"Имя: {user_data['name']}, "
                             f"Телефон: {user_data['phone']}, "
                             f"E-mail: {user_data['email']}, "
                             f"Статус: {user_data['status']}, "
                             f"Опыт работы: {user_data['experience']}, "
                             f"Навыки: {user_data['skills']}, "
                             f"Пожелания: {user_data['wishes']}, "
                             f"Желаемая зарплата: {user_data['salary']}. "
                             f"Сгенерируй резюме в удобном для работодателя формате.")
    ]

    # Отправляем данные на генерацию резюме
    generated_resume = await ask_gigachat(messages)
```

```
    # Генерация PDF
    pdf_filename = f"{user_data['name']}_resume.pdf"
    create_pdf(generated_resume, pdf_filename)

    # Отправляем готовое резюме пользователю
    await message.answer(generated_resume)
    await message.answer_document(types.InputFile(pdf_filename))

    await message.answer(
        "Если хотите, вы можете отредактировать резюме или оценить его. Используйте кнопки для навигации.")
    await ResumeForm.finalize.set()


# Хендлер для оценки резюме
@dp.message_handler(lambda message: message.text == "Оценить резюме", state="*")
async def evaluate_resume(message: types.Message, state: FSMContext):
    await message.answer("Пожалуйста, отправьте ваше резюме для оценки.")
    await ResumeForm.evaluation.set()


@dp.message_handler(state=ResumeForm.evaluation)
async def process_evaluation(message: types.Message, state: FSMContext):
    user_resume = message.text

    messages = [
        SystemMessage(content="Ты бот, который оценивает резюме."),
        HumanMessage(content=f"Резюме для оценки: {user_resume}")
    ]

    evaluation_response = await ask_gigachat(messages)
    await message.answer(f"Оценка вашего резюме:\n\n{evaluation_response}\n\n"
                         "Хотите, чтобы я предложил исправления? Ответьте 'Да' или 'Нет'.")
    await ResumeForm.finalize.set()

                                                # Хендлер для обработки ответа на предложение исправлений
@dp.message_handler(lambda message: message.text in ['Да', 'Нет'], state=ResumeForm.finalize)
async def handle_corrections_response(message: types.Message, state: FSMContext):
    if message.text == 'Да':
        await message.answer("Отлично! Какие исправления вы хотите внести? Пожалуйста, укажите.")
        await ResumeForm.edit.set()  # Вернуться в режим редактирования
    else:
        await message.answer("Хорошо! Если вам понадобится помощь, просто напишите мне.")
        await state.finish()  # Завершаем состояние


# Логика нового подхода для редактирования резюме
def get_edit_menu():
    markup = ReplyKeyboardMarkup(resize_keyboard=True)
    name_button = KeyboardButton("Имя")
    phone_button = KeyboardButton("Телефон")
    email_button = KeyboardButton("Email")
    status_button = KeyboardButton("Статус")
    experience_button = KeyboardButton("Опыт работы")
    skills_button = KeyboardButton("Навыки")
    wishes_button = KeyboardButton("Пожелания")
    salary_button = KeyboardButton("Зарплата")
    cancel_button = KeyboardButton("Отмена")

    markup.add(name_button, phone_button, email_button)
    markup.add(status_button, experience_button, skills_button)
    markup.add(wishes_button, salary_button, cancel_button)
    return markup
```

```
@dp.message_handler(lambda message: message.text == "Редактировать", state=ResumeForm.finalize)
async def initiate_edit(message: types.Message, state: FSMContext):
    await message.answer("Какое поле вы хотите отредактировать? Выберите одно из предложенных ниже.",
                         reply_markup=get_edit_menu())
    await ResumeForm.edit.set()


@dp.message_handler(state=ResumeForm.edit)
async def process_edit_choice(message: types.Message, state: FSMContext):
    field_to_edit = message.text.lower()

    if field_to_edit in ['имя', 'телефон', 'email', 'статус', 'опыт работы', 'навыки', 'пожелания', 'зарплата']:
        await state.update_data(edit_field=field_to_edit)
        await message.answer(f"Введите новое значение для поля '{field_to_edit}':")
        await ResumeForm.waiting_for_new_value.set()
    elif field_to_edit == "отмена":
        await message.answer("Редактирование отменено.")
        await ResumeForm.finalize.set()
    else:
        await message.answer("Неверное поле. Пожалуйста, выберите одно из предложенных.")


@dp.message_handler(state=ResumeForm.waiting_for_new_value)
async def update_field(message: types.Message, state: FSMContext):
    user_data = await state.get_data()
    edit_field = (await state.get_data()).get("edit_field")
    new_value = message.text

    if edit_field:
        if edit_field == 'имя':
            await state.update_data(name=new_value)
        elif edit_field == 'телефон':
            await state.update_data(phone=new_value)
        elif edit_field == 'email':
            await state.update_data(email=new_value)
        elif edit_field == 'статус':
            await state.update_data(status=new_value)
        elif edit_field == 'опыт работы':
            await state.update_data(experience=new_value)
        elif edit_field == 'навыки':
            await state.update_data(skills=new_value)
        elif edit_field == 'пожелания':
            await state.update_data(wishes=new_value)
        elif edit_field == 'зарплата':
            await state.update_data(salary=new_value)

        await message.answer(f"Поле '{edit_field}' обновлено на '{new_value}'.")

        # Подготовка данных для обновленного резюме
        updated_user_data = await state.get_data()

        promptone = """
            Вы — бот, формирующий резюме на основе предоставленных данных.
            Используйте только информацию, предоставленную пользователем, и избегайте добавления данных, которых нет в вводе.
            Не создавайте и не выдумывайте факты, не вносите оценочные суждения или приукрашивания.
            Ваша задача — просто структурировать предоставленные данные в ясном и убедительном формате, подходящем для резюме.
            Резюме должно быть разделено на: контактная информация, опыт работы, навыки, пожелания и краткое описание кандидата, основанное на ключевых компетенциях и целях.
            """
```

```
        messages = [
            SystemMessage(content=promptone),
            HumanMessage(content=f"Имя: {updated_user_data.get('name', 'не указано')}, "
                         f"Телефон: {updated_user_data.get('phone', 'не указано')}, "
                         f"E-mail: {updated_user_data.get('email', 'не указано')}, "
                         f"Статус: {updated_user_data.get('status', 'не указано')}, "
                         f"Опыт работы: {updated_user_data.get('experience', 'не указано')}, "
                         f"Навыки: {updated_user_data.get('skills', 'не указано')}, "
                         f"Пожелания: {updated_user_data.get('wishes', 'не указано')}, "
                         f"Желаемая зарплата: {updated_user_data.get('salary', 'не указано')}. "
                         f"Сгенерируй резюме в удобном для работодателя формате.")
        ]

        # Генерация обновленного резюме
        generated_resume = await ask_gigachat(messages)

        # Генерация нового PDF
        pdf_filename = f"{updated_user_data['name']}_resume.pdf"
        create_pdf(generated_resume, pdf_filename)

        # Отправляем готовое обновленное резюме пользователю
        await message.answer(generated_resume)
        await message.answer_document(types.InputFile(pdf_filename))

        await ResumeForm.finalize.set()
    else:
        await message.answer("Не удалось определить поле для обновления. Пожалуйста, повторите попытку.")

# Хендлер для поиска вакансий
@dp.message_handler(lambda message: message.text == "Вакансия", state=ResumeForm.finalize)
async def search_vacancies(message: types.Message, state: FSMContext):
    user_data = await state.get_data()

    # Используем навыки и пожелания для поиска вакансий
    skills = user_data.get("skills", "")

    # Формируем запрос для поиска вакансий
    query = f"{skills}"

    # Поиск вакансий на hh.ru (например)
    url = f"https://rabota.ykt.ru/jobs?text={query}&rcategoriesIds=&salaryMin=&salaryMax=&period=ALL"

    await message.answer(f"Вот список вакансий, подходящих под ваше резюме: \n{url}")

def fetch_vacancies(query):
    url = f"https://rabota.ykt.ru/jobs?text={query}&rcategoriesIds=&salaryMin=&salaryMax=&period=ALL"
    response = requests.get(url)
    soup = BeautifulSoup(response.content, 'html.parser')

    vacancies = []
    for vacancy in soup.find_all('a', {'data-qa': 'serp-item__title'}, limit=5):  # Получаем первые 5 вакансий
        title = vacancy.text
        link = vacancy['href']
        vacancies.append(f"{title}: {link}")

    return vacancies

@dp.message_handler(lambda message: message.text == "Вакансия", state=ResumeForm.finalize)
async def search_vacancies(message: types.Message, state: FSMContext):
    user_data = await state.get_data()

    # Используем навыки и пожелания для поиска вакансий
    skills = user_data.get("skills", "")
```

```
    # Используем навыки и пожелания для поиска вакансий
    skills = user_data.get("skills", "")

    # Формируем запрос для поиска вакансий
    query = f"{skills}"

    # Получаем список вакансий
    vacancies = fetch_vacancies(query)

    # Выводим вакансии пользователю
    if vacancies:
        await message.answer("Вот несколько подходящих вакансий:\n" + "\n".join(vacancies))
    else:
        await message.answer("К сожалению, подходящих вакансий не найдено.")

@dp.message_handler(state=ResumeForm.finalize)
async def finalize_resume(message: types.Message, state: FSMContext):
    await message.answer("Ваше резюме готово! Пожалуйста, оставьте отзыв о нашей работе.")
    await ResumeForm.feedback.set()

@dp.message_handler(state=ResumeForm.feedback)
async def collect_feedback(message: types.Message, state: FSMContext):
    feedback = message.text

    # Логируем или сохраняем отзыв
    logging.info(f"Отзыв от {message.from_user.id}: {feedback}")

    # Можно отправить отзыв в чат или на email админа, если требуется

    await message.answer("Спасибо за ваш отзыв! Если возникнут дополнительные вопросы, пишите!")
    await state.finish()  # Завершаем состояние

# Запуск бота
if __name__ == '__main__':
    executor.start_polling(dp, skip_updates=True)
```