  
Автономная некоммерческая организация   
Дополнительного профессионального образования

Компьютерная Академия «ТОП» филиал «Академия ТОП Уфа»

Дипломная работа  
по курсу «Веб-разработка на Python»  
  
«Разработка Telegram-бота для автоматизации системы заказа еды»

Выполнил студент Компьютерной Академии ТОП   
Мухамадеев Айдар Илдарович /\_\_\_\_\_\_\_\_\_\_\_\_  
   
Дипломная работа допущена к защите и проверена на объем заимствования:

Директор филиала  
 АНО ДПО «Академия ТОП»   
 Игнатьева Азалия Фаритовна\ \_\_\_\_\_\_\_\_\_\_\_\_\_

Рук. учебной части филиала   
 АНО ДПО «Академия   
 Фатхинурова Светлана Форагатовна\\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Уфа, 2024г.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

[Глава 1. Анализ предметной области 4](#_Toc183007164)

[1.1. Обзор рынка Telegram-ботов …. 4](#_Toc183007165)

[1.2. Функциональные требования к чат-боту 5](#_Toc183007166)

[Глава 2. обзор технологий 6](#_Toc183007167)

2.[1. Архитектура разрабатываемого программного обеспечения 6](#_Toc183007169)

2.[2. Функциональные требования 14](#_Toc183007169)

[Глава 3. Реализация 17](#_Toc183007172)

[3.1. Структура проекта 17](#_Toc183007173)

[3.2. Реализация функционала 20](#_Toc183007174)

[3.3. Тестирование и отладка 23](#_Toc183007174)

ЗАКЛЮЧЕНИЕ

**ВВЕДЕНИЕ**

В условиях стремительного развития технологий и растущей популярности онлайн-сервисов, автоматизация процессов становится всё более актуальной. Настоящая дипломная работа посвящена разработке чат-бота для автоматизации системы заказа еды, призванного повысить эффективность и удобство взаимодействия между клиентами и заведениями общественного питания.

В работе рассматриваются вопросы проектирования, разработки и внедрения интеллектуальной системы на основе современных технологий обработки естественного языка и машинного обучения, позволяющей автоматизировать процесс приема заказов, обработки платежей и предоставления информации о меню и статусе заказа. Актуальность выбранной темы обусловлена потребностью в оптимизации работы служб доставки и ресторанов, а также повышении удовлетворенности клиентов за счет быстрого и удобного интерфейса. В рамках исследования будут проанализированы существующие решения в данной области, описаны этапы разработки собственного чат-бота, представлены результаты тестирования и оценки его эффективности.

Бот должен обеспечивать простой интерфейс взаимодействия с пользователями, предоставляя информацию о меню, а также возможность выбора блюд и оформления заказа в режиме реального времени. Данная система позволит повысить эффективность процесса заказа, уменьшить время ожидания ответов и улучшить общее качество обслуживания клиентов. Бот реализован на языке программирования Python с использованием фреймворка Django и СУБД PostgreSQL.

**ГЛАВА 1. АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ**

### **Обзор рынка Telegram-ботов**

Анализ рынка сервисов доставки еды и онлайн-платформ для заказа выявил широкое распространение систем, частично или полностью использующих чат-ботов для автоматизации взаимодействия с клиентами. Эти системы демонстрируют различные подходы к реализации, отличающиеся по уровню сложности, функциональности и используемым технологиям. Можно выделить несколько категорий существующих решений:

* Многие приложения для заказа еды используют чат-ботов для упрощения процесса заказа, предоставления информации о статусе доставки и решения простых вопросов. Эти боты, как правило, обладают ограниченным функционалом и опираются на заранее запрограммированные сценарии. Недостатком таких решений часто является неспособность обрабатывать нестандартные запросы или сложные ситуации, что требует вмешательства операторов.
* Чат-боты, работающие в мессенджерах: Некоторые сервисы предлагают взаимодействие с клиентами через популярные мессенджеры, такие как Telegram, WhatsApp или Facebook Messenger. Это позволяет пользователям заказывать еду в привычной для них среде.
* Гибридные системы: Некоторые сервисы используют комбинацию автоматизированных и ручных процессов. Чат-бот отвечает на вопросы и обрабатывает стандартные заказы, а операторы принимают участие в решении сложных проблем или обработке нестандартных запросов. Такой подход позволяет обеспечить баланс между автоматизацией и качеством обслуживания.

Анализ существующих решений показал, что, несмотря на значительный прогресс в области разработки чат-ботов, многие системы страдают от ограниченного функционала, проблем с распознаванием естественного языка, недостаточно интуитивного интерфейса, длительного времени ожидания ответа и отсутствия возможности обработки сложных или нестандартных запросов. Это указывает на необходимость разработки более совершенных и гибких систем, которые бы обеспечивали высокое качество обслуживания и удовлетворяли потребности как пользователей, так и заведений общественного питания.

**1.2. Функциональные требования к чат-боту.**

Разрабатываемый чат-бот должен обеспечивать следующие основные функции:

* Прием заказов: автоматическое распознавание блюд и напитков из меню, обработка количества и модификаций заказов.
* Обработка платежей: интеграция с платежными системами для безопасной и удобной оплаты.
* Управление доставкой: интеграция с сервисами доставки, отслеживание статуса заказа, уведомления о времени доставки.
* Предоставление информации: доступ к меню, информации о ценах, акциях и специальных предложениях.
* Обработка запросов: ответы на часто задаваемые вопросы, решение проблемных ситуаций (например, изменение или отмена заказа).
* Удобство использования: интуитивный и понятный интерфейс, быстрая скорость ответа.
* Совместимость: работа с различными платформами и устройствами (мобильные приложения, веб-сайты, мессенджеры).

**ГЛАВА 2. ОБЗОР ТЕХНОЛОГИЙ**

**2.1. Архитектура разрабатываемого программного обеспечения**

Архитектура разрабатываемого Telegram-бота для автоматизации системы заказа еды будет основана на клиент-серверной модели с использованием многоуровневого подхода и фреймворка Django. Это обеспечит масштабируемость, поддерживаемость и гибкость системы. Основные компоненты архитектуры:

1. Telegram Bot API: Интерфейс программирования приложений (API) Telegram, предоставляющий доступ к функциональности Telegram для создания ботов. Бот будет использовать этот API для взаимодействия с пользователями, приема сообщений и отправки ответов, включая кнопки и инлайн-меню для улучшения пользовательского опыта (UX).

2. Серверная часть (Django): Серверная часть будет реализована с использованием фреймворка Django на языке Python и будет отвечать за обработку запросов от бота, взаимодействие с базой данных PostgreSQL и внешними сервисами. Она будет состоять из следующих модулей:

Модуль обработки сообщений (views.py): Этот модуль, реализованный с использованием Django views, будет получать сообщения от Telegram Bot API (через библиотеки python-telegram-bot или aiogram), распознавать намерения пользователя (с использованием NLP-модуля, если таковой применяется) и определять дальнейшие действия, обрабатывая навигацию по инлайн-меню и кнопкам.

Модуль управления заказами (models.py, forms.py): Этот модуль, используя Django models и forms, будет отвечать за создание, обработку и отслеживание заказов. Он будет взаимодействовать с базой данных PostgreSQL для хранения информации о заказах и меню.

Модуль интеграции с платежными системами: Этот модуль обеспечит безопасную обработку платежей с использованием API выбранной платежной системы. Интеграция будет осуществлена через соответствующие библиотеки Python.

Модуль интеграции с сервисами доставки: Этот модуль будет взаимодействовать с API выбранного сервиса доставки (например, Delivery Club API) для организации доставки заказов. Интеграция будет осуществлена через соответствующие библиотеки Python.

Модуль базы данных (PostgreSQL): База данных PostgreSQL будет хранить информацию о пользователях, меню, заказах, и других необходимых данных. Django ORM обеспечит удобное взаимодействие с базой данных.

Модуль логирования: Этот модуль будет регистрировать все действия бота и важные события для отладки и мониторинга.

3. Клиентская часть (Telegram-клиент): Это приложение пользователя, через которое он взаимодействует с ботом. В данном случае, это сам Telegram-клиент, взаимодействующий с ботом через удобный интерфейс, реализованный с помощью кнопок и инлайн-меню.

4. Взаимодействие компонентов:

Пользователь отправляет сообщение или взаимодействует с кнопками/инлайн-меню в Telegram.

Telegram Bot API передает информацию на сервер Django.

Django обрабатывает запрос, используя соответствующие модули и взаимодействуя с базой данных PostgreSQL.

Сервер отправляет ответ (включая кнопки и инлайн-меню) через Telegram Bot API обратно пользователю.

Данная архитектура, основанная на Django, позволяет эффективно использовать возможности этого фреймворка для разработки надежного, масштабируемого и поддерживаемого Telegram-бота. Использование PostgreSQL в качестве базы данных обеспечивает высокую производительность и надежность хранения данных. Выбор библиотек python-telegram-bot или aiogram обеспечивает удобное взаимодействие с Telegram Bot API. Применение кнопок и инлайн-меню значительно улучшает UX, делая взаимодействие с ботом интуитивно понятным.

Таким образом, выбранные технологии создают надёжную и эффективную основу для разработки бота, обеспечивая его функциональность и высокое качество пользовательского опыта.

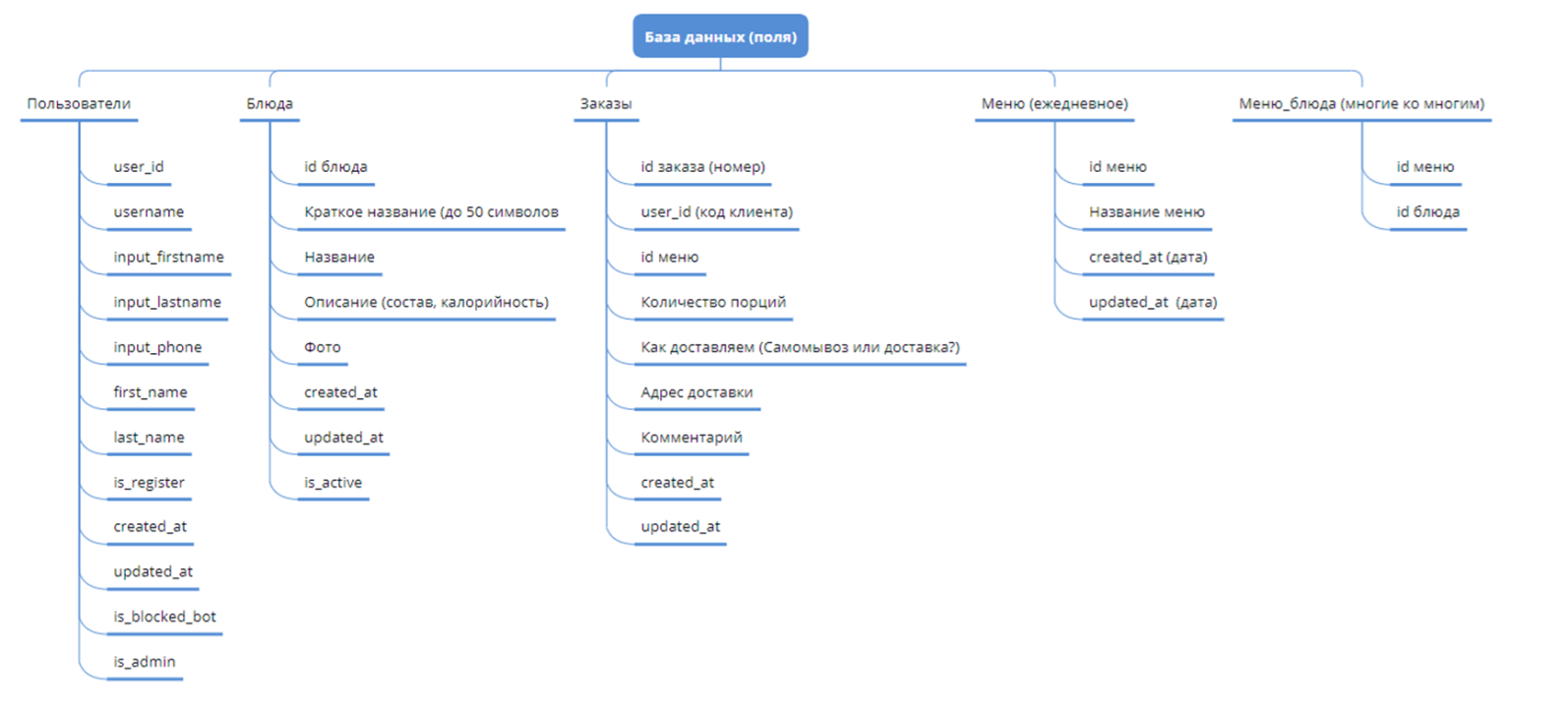
****

Рисунок 1. Схема базы данных

Схема на изображении описывает структуру базы данных, состоящей из нескольких таблиц:

Пользователи: Таблица содержит информацию о пользователях приложения, включая их идентификатор (user\_id), имя пользователя (username), имя и фамилию ( input\_firstname, input\_lastname, first\_name, last\_name), номер телефона (input\_phone), флаги регистрации (is\_register), активности (is\_active), блокировки ботом (is\_blocked\_bot) и администраторских прав (is\_admin), а также метки времени создания и обновления записи (created\_at, updated\_at).

Блюда: Таблица описывает каждое блюдо в меню, с полями для идентификатора (id блюда), краткого и полного названия (Краткое название, Название), описания ( Описание), пути к изображению (Фото) и метками времени создания и обновления записи (created\_at, updated\_at). Поле is\_active вероятно указывает на доступность блюда в меню.

Меню (ежедневное): Таблица содержит информацию о ежедневном меню, включая идентификатор (id меню), название (Название меню) и метки времени создания и обновления (created\_at, updated\_at).

Заказы: Таблица хранит информацию о заказах, включая идентификатор (id заказа), идентификатор пользователя (user\_id), идентификатор меню (id меню), количество порций (Количество порций), способ доставки (Как доставляем), адрес доставки (Адрес доставки), комментарии (Комментарий) и метки времени создания и обновления (created\_at, updated\_at).

Меню\_блюда: Это связующая таблица, реализующая отношение "многие ко многим" между таблицами "Меню (ежедневное)" и "Блюда". Она содержит идентификаторы меню (id меню) и блюда (id блюда), позволяя одному меню включать в себя множество блюд.

### 1. **handlers**

Приветствует пользователя в клиентском разделе и показывает правила.

Отправляет текстовые сообщения и клавиатуру.

Устанавливает состояние ожидания обработки правил.

**client\_handling\_rules:**

Обрабатывает разрешение пользователя на получение сообщений.

Если пользователь согласен, вызывается функция user\_registration для регистрации, и состояние изменяется на ожидание информации о заказе.

Если пользователь отказывается, состояние очищается, и отправляется сообщение об отказе.

**client\_msg\_before\_start:**

Отправляет приветственное сообщение перед началом взаимодействия с ботом.

**client\_if\_order:**

Запрашивает у клиента, хочет ли он заказывать еду на завтра.

Обрабатывает ответ: если «Да», то переходит к началу заказа; если «Нет», обрабатывает отказ.

**client\_cancel\_order:**

Обрабатывает отказ от заказа, отправляя соответствующее сообщение.

**client\_start\_order:**

Если пользователь согласен заказывать, бот запрашивает количество порций.

**client\_count\_of\_serv:**

Обрабатывает количество порций и проверяет тип оплаты.

Вызывает проверку количества порций, и, если оно больше 100, повторно запрашивает количество.

check\_count\_of\_servings:

Валидация введенного количества порций с использованием регулярного выражения.

Если количество неправильно, отправляет уведомление и возвращается к началу заказа.

**client\_sel\_payment:**

Обрабатывает выбор способа оплаты (картой или наличными).

Обновляет соответствующее состояние и включает клавиатуру для выбора доставки.

**client\_payment\_cash:**

Производит валидацию введенной суммы при оплате наличными.

Обрабатывает ситуации, если введенное значение не соответствует ожиданиям (используя регулярные выражения).

**client\_check\_order:**

Описание: Сохранение заказа в базе данных, если все введенные данные верны.

Если пользователь подтверждает правильность данных, они собираются в словарь order\_data и отправляются в функцию add\_order для добавления заказа в базу данных.

Если добавление прошло успешно (код статуса 201), отправляется подтверждение заказа.

Если возникла ошибка при добавлении, отправляется сообщение об ошибке, с возможностью попробовать снова.

Если пользователь хочет исправить данные, возвращается к началу процесса создания заказа.

Эти функции взаимодействуют с пользователем, позволяя ему сделать полноценный заказ, проверить введенные данные и обработать финальные шаги, связанные с оформлением заказа. Они обеспечивают последовательность и контроль за процессом, а также поддержку состояния пользователя в процессе взаимодействия с ботом.

1. keyboards

Каждая функция создает кнопки для различных операций администратора, что упрощает управление ботом и взаимодействие с пользователями. Клавиатуры обеспечивают удобный интерфейс для выполнения административных задач в рамках бота. Клавиатуры, возвращающие ReplyKeyboardMarkup, автоматически настраиваются под размеры экрана устройства, так как задан параметр resize\_keyboard=True, что улучшает пользовательский опыт.

Клавиатуры Администратора

admin\_check\_accept\_kb():

Создаёт клавиатуру с одной кнопкой (пока без текста).

Возвращает разметку клавиатуры с одной кнопкой.

admin\_main\_menu\_kb():

Создаёт главное меню с кнопками для работы с меню, блюдами и заказами.

Возвращает клавиатуру с тремя кнопками.

Работа с заказами:

admin\_work\_with\_orders\_kb():

Создаёт клавиатуру с кнопками для отображения списка заказов и для возврата в главное меню.

admin\_choose\_orders\_kb():

Создаёт клавиатуру с кнопками для выбора заказа по дате (сегодня, завтра, выбор по дате) и возвратом в главное меню.

Работа с блюдами:

admin\_work\_with\_dishes\_kb():

Создаёт клавиатуру для управления блюдами: просмотра списка, добавления, изменения и удаления блюд.

admin\_add\_dish\_noimage\_kb():

Клавиатура для выбора блюда без фото.

admin\_add\_dish\_check\_kb():

Подтверждение корректности введённых данных с выбором между подтверждением или исправлением.

admin\_start\_repair\_dish\_kb():

Клавиатура для выбора параметров, которые нужно изменить (название, краткое название, описание, фотография).

admin\_remove\_dish\_choose\_kb():

Подтверждение удаления блюда с выбором между подтверждением и отменой.

Работа с меню:

admin\_work\_with\_menus\_kb():

Создаёт клавиатуру для управления меню: создание нового меню, удаление, просмотр созданных меню и массовая рассылка.

admin\_add\_menus\_confirm\_kb():

Подтверждение правильности данных о меню.

admin\_remove\_menu\_choose\_kb():

Подтверждение удаления меню: подтверждение или отмена.

admin\_confirm\_send\_kb():

Подтверждение отправки меню с вариантами «Да» и «Нет».

1. phrases

Данный раздел определяет р фразы, используемые в взаимодействии с пользователем для Telegram-бота, который может принимать заказы на еду. Эти фразы представляют собой тексты, которые бот отправляет пользователю на различных этапах процесса взаимодействия, от регистрации до оформления заказа. Рассмотрим на каждую секцию и её назначение.

Регистрация Пользователя

**check\_registration:**

CHECK\_REG\_1: Сообщение для зарегистрированных пользователей с инструкциями по переходу к заказам.

CHECK\_REG\_2: Уведомление для незарегистрированных пользователей с просьбой зарегистрироваться.

**start\_registration:**

START\_REG\_1: Направляющее сообщение для начала процесса регистрации.

START\_REG\_2: Запрос имени пользователя.

**input\_firstname:**

INPUT\_NAME\_1: Ошибка, если имя введено неверно (допускаются только буквы и тире).

INPUT\_NAME\_2: Запрос фамилии после введения имени.

**input\_lastname:**

INPUT\_LASTNAME\_1: Ошибка, если фамилия введена неверно.

INPUT\_LASTNAME\_2: Запрос номера телефона.

**input\_phone:**

INPUT\_PHONE\_1: Ошибка, если телефон введен в неверном формате.

INPUT\_PHONE\_2: Подтверждение правильности ввода номера телефона.

**confirm\_registration:**

CONFIRM\_REG\_1: Сообщение об успешной регистрации и шаге оформления заказа.

CONFIRM\_REG\_2: Предложение повторить процесс, если возникла ошибка.

CONFIRM\_REG\_ERR: Сообщение об ошибке в процессе регистрации.

Основной Раздел Пользователя

**enter\_client\_section:**

ENTER\_CLIENT\_1: Приветственное сообщение с указанием времени для раздумий о заказе.

ENTER\_CLIENT\_2: Запрос согласия с правилами.

**client\_handling\_rules:**

CLIENT\_ACCEPT: Сообщение о том, что пользователь исключён из рассылки.

**client\_msg\_before\_start:**

CLIENT\_START\_MSG: Сообщение о том, что меню будет отправлено пользователю.

**client\_cancel\_order:**

CL\_CANCEL: Информация о том, что заказ не будет оформлен.

Заказ

**client\_start\_order:**

CL\_START\_ORDER: Запрос на указание количества порций.

**client\_count\_of\_serv:**

CL\_COUNT: Сообщение об ограничении на количество порций.

CL\_COUNT\_2: Подтверждение о количестве порций и запрос на выбор способа оплаты.

**check\_count\_of\_servings:**

Этот набор фраз предоставляет четкий и последовательный пользовательский интерфейс в боте. Они помогают пользователю на каждом этапе, от регистрации до оформления заказа, предоставляя нужную информацию и поддержку. Правильная организация текстов способствует лучшему взаимодействию между ботом и пользователем, облегчая процесс выполнения заказа.

**2.2. Функциональные требования**

Функциональные требования для чат-бота доставки еды могут быть разделены на несколько ключевых категорий, включая регистрацию пользователей, процесс заказа, управление меню, обработку платежей и взаимодействие с пользователями. Ниже приводится детализированный список функциональных требований.

1. Регистрация и аутентификация пользователей

Регистрация пользователя:

* Пользователь должен иметь возможность зарегистрироваться через бота, вводя свое имя, фамилию и номер телефона.
* Бот должен проверять корректность введённых данных (имя и фамилия должны состоять только из букв и тире, номер телефона должен соответствовать формату).
* Если пользователь уже зарегистрирован, бот должен уведомить его об этом.

Аутентификация пользователя:

* Пользователь должен иметь возможность войти в систему (если требуется) и получить доступ к своим данным и истории заказов.

2. Процесс заказа

* Создание заказа:
* Пользователь может в любой момент инициировать заказ, запросив меню на ближайшую дату.
* Бот должен спрашивать количество порций, которые пользователь хочет заказать.
* Пользователь должен иметь возможность указать тип получения заказа (доставка или самовывоз).

Выбор блюд:

* Бот должен предоставлять пользователю меню с доступными блюдами, включая описание и цену.
* Пользователь должен иметь возможность выбрать одно или несколько блюд для заказа.

3. Оплата

Выбор способа оплаты:

* Пользователь должен иметь возможность выбрать метод оплаты: наличными или картой.

Обработка платежей:

* При выборе оплаты по карте бот должен обеспечить безопасность и конфиденциальность операций.
* Если пользователь выбирает оплату наличными, бот должен запросить сумму, с которой необходима сдача.

4. Доставка

Адрес доставки:

* При выборе доставки бот должен запросить и подтвердить адрес доставки.
* Бот должен поступить с ограничениями на места доставки (например, только в пределах определённого жилого комплекса).

Статус доставки:

* Пользователь должен иметь возможность отслеживать статус своего заказа: ожидается ли доставка, в процессе или завершена.

5. Управление меню

* Получение актуального меню:
* Бот должен гарантировать, что пользователи получают актуальное, обновлённое меню.
* Обновление меню:
* Если администратор или менеджер добавляет или изменяет меню, бот должен сообщить об изменениях всем пользователям.
* Поддержка пользователей

6. Административные функции

Управление заказами:

* Администраторы должны иметь возможность просматривать список всех заказов, управлять ими и изменять статус.
* Управление блюдами:
* Администраторы должны иметь возможность добавлять, изменять и удалять блюда из меню.
* Статистика и отчёты:
* Бот должен предоставлять статистику по заказам, отзывам и продажам, доступную для администраторов.

7. Пользовательский интерфейс

Удобный интерфейс:

* Чат-бот должен иметь интуитивно понятный интерфейс с кнопками для выбора определённых опций.

Информация и уведомления:

* Бот должен отправлять уведомления пользователю о статусе заказа, специальных предложениях и акциях.

Эти функциональные требования обеспечивают создание эффективного, удобного и безопасного чат-бота для доставки еды. Они помогают пользователям легко заказывать еду и контролировать свои заказы, а также позволяют администраторам эффективно управлять процессом.

# **ГЛАВА 3. РЕАЛИЗАЦИЯ**

## **3.1. Структура проекта**

Проект был организован с учётом принципов, заложенных в Django, что обеспечило структурированность и удобство разработки. Основная структура проекта выглядит следующим образом:

food\_delivery\_bot/

│

├── bot\_app/

│ ├── \_\_init\_\_.py

│ ├── bot\_create.py # Инициализация бота и установка соединения с API Telegram

│ ├── data\_exchanger.py # Функции для работы с базой данных: добавление заказов, регистрация пользователей и т.д.

│ ├── handlers/

│ │ ├── \_\_init\_\_.py

│ │ ├── client\_handlers.py # Обработчики для пользовательских команд (регистрация, заказ и т.д.)

│ │ ├── admin\_handlers.py # Обработчики для административных команд

│ │ ├── utils.py # Утилиты: функции проверки, обработки ошибок и т.д.

│ │ └── middlewares.py # Middleware для обработки сообщений (например, логирование)

│ ├── keyboards/

│ │ ├── \_\_init\_\_.py

│ │ ├── user\_kb.py # Клавиатуры для пользователей

│ │ └── admin\_kb.py # Клавиатуры для администраторов

│ ├── phrases/

│ │ ├── \_\_init\_\_.py

│ │ └── user\_phrases.py # Фразы, используемые в общении с пользователями

│ ├── states.py # Определение состояний для FSM (Finite State Machine)

│ └── models.py # Определения моделей данных (если используется ORM)

│

├── config.py # Конфигурационный файл с настройками бота (токены, базы данных и т.д.)

├── requirements.txt # Список зависимостей Python

└── main.py # Основной файл для запуска бота

Описание компонентов

**bot\_app/:** Основная папка приложения, содержащая все компоненты чат-бота.

bot\_create.py: Содержит код для инициализации бота и настройки соединения с API Telegram с использованием aiogram.

data\_exchanger.py: Содержит функции для работы с данными, такие как регистрация пользователей, добавление заказов в базу данных и их получение.

**handlers/:**

client\_handlers.py: Содержит обработчики команд и сообщений пользователей, такие как процесс регистрации и оформление заказа.

admin\_handlers.py: Содержит обработчики команд для администраторов (например, управление меню, просмотр заказов).

utils.py: Вспомогательные функции, проверки и обработка различных данных.

middlewares.py: Опционально содержит обработчики промежуточных слоёв, такие как логирование запросов.

**keyboards/:**

user\_kb.py: Определяет клавиатуры для взаимодействия с пользователями, включая кнопки для выбора меню, подтверждений и т.д.

admin\_kb.py: Определяет клавиатуры для администраторов, чтобы они могли управлять заказами и блюдами.

**phrases/:**

user\_phrases.py: Содержит все фразы, которые будут отправляться пользователям на разных этапах взаимодействия.

states.py: Определение различных состояний машины состояний (FSM), что помогает контролировать поток сообщений и взаимодействия с пользователями.

models.py: Если используется ORM (например, SQLAlchemy или Django ORM), здесь будут определены модели данных для хранения информации о пользователях и заказах.

config.py: Файл с конфигурацией проекта, включая параметры подключения к базе данных и токены API.

requirements.txt: Содержит список всех зависимостей для проекта (например, aiogram, psycopg2 для работы с PostgreSQL, и другие необходимые библиотеки).

main.py: Основной файл, в котором происходит запуск бота, инициализируются обработчики и запускается цикл обработки обновлений..

## **3.2. Реализация функционала**

Чат-боты стали неотъемлемой частью современного бизнеса, предоставляя удобные и эффективные способы взаимодействия с клиентами.

Регистрация пользователей — это ключевой шаг в любом приложении, основанном на взаимодействии с пользователем. В контексте чат-бота для доставки еды регистрация позволяет закрепить за пользователем информацию, необходимую для выполнения его заказов.

Сбор информации: Бот собирает имя, фамилию и номер телефона через последовательные запросы к пользователю. Этот процесс подразумевает использование состояний, чтобы контролировать, на каком этапе регистрации находится пользователь.

Валидация данных: Важно убедиться, что пользователи вводят данные в правильном формате (например, имя и фамилия должны содержать только буквы, а номер телефона — соответствовать определенному шаблону). Валидация позволяет предотвратить ошибки и обеспечивает корректное функционирование последующих этапов.

Хранение данных: После успешной регистрации данные пользователя сохраняются в базе данных, что позволяет системе идентифицировать его на следующих этапах взаимодействия.

2. Создание заказа

Создание заказа является центральным элементом функционала чат-бота. Этот процесс должен быть интуитивным и минимально затрудняющим пользователя. Запрос меню: Бот должен предоставлять пользователю актуальное меню с возможностью выбора блюд. Эта информация может быть динамически загружена из базы данных.

Определение количества порций: После выбора блюда бот запрашивает, сколько порций пользователь хочет заказать. Важно установить пределы, чтобы предотвратить избыточные заказы.

Выбор способа получения заказа: Пользователь может выбрать удобный способ получения заказа — самовывоз или доставка. Выбор должен поощрять пользователя, и бот должен объяснять, что именно включает каждый из вариантов.

3. Оплата заказа

Обработка оплаты — это важный аспект работы чат-бота, так как она напрямую связана с удовлетворённостью пользователей и безопасностью транзакций. Выбор метода оплаты: Бот должен предложить несколько способов оплаты (наличные, карта) и четко информировать пользователя о каждом процессе.

Валидация платежных данных: При выборе наличного расчета бот запрашивает сумму, с которой требуется сдача. Важно обеспечить, чтобы сумма была введена корректно, так как это может повлиять на процесс выполнения заказа.

4. Организация доставки

Организация доставки — ключевой шаг в завершении процесса заказа, который непосредственно влияет на удовлетворенность клиента.

Запрос адреса доставки: Бот должен запросить у пользователя адрес, по которому будет произведена доставка. Этот процесс может включать проверку на корректность введенного адреса. Подтверждение адреса: Перед окончательной отправкой заказа важно подтвердить адрес, чтобы избежать ошибок и недоразумений.

5. Подтверждение и управление заказом

Подтверждение заказа включает в себя предоставление пользователю всей информации о заказе для окончательной проверки. Проверка данных: Бот должен сформировать итоговое сообщение с деталями заказа и запросить подтверждение от пользователя. Это помогает убедиться, что информация верна до того, как заказ будет окончательно оформлен и передан в систему обработки.

6. Обработка ошибок и исключений

Эффективная обработка ошибок является важной частью любого программного обеспечения. Отправка сообщений об ошибках: Если бот сталкивается с проблемами (например, неверный ввод данных), он должен сообщать пользователю о проблеме и предлагать шаги для её устранения.

Реализация функционала чат-бота доставки еды требует тщательного проектирования и интеграции различных компонентов, начиная от регистрации пользователей и заканчивая обработкой заказов и платежей. Четкая структура интерфейса и продуманное взаимодействие с пользователями позволяют повысить общий уровень удовлетворённости и улучшить пользовательский опыт. Эти элементы являются основой для успешного функционирования чат-бота в сфере доставки еды и позволяют дифференцировать сервис на фоне конкурентов.

## **3.3. Тестирование и отладка**

1. Подходы к тестированию чат-бота

1.1. Функциональное тестирование

Функциональное тестирование направлено на проверку того, выполняет ли бот все предусмотренные функции. Ключевые аспекты функционального тестирования включают:

* Регистрация пользователей: Тестирование всех этапов регистрации, включая ввод имени, фамилии и телефона, чтобы убедиться, что данные валидируются правильно.
* Создание заказа: Проверка возможности создания заказа от выбора блюда до выбора способа получения.
* Оплата: Тестирование различных методов оплаты (наличные, карта), включая валидацию данных о платежах и подсчет сдачи.
* Доставка: Проверка процесса запроса адреса доставки и его валидации.

1.2. Нефункциональное тестирование

* Нефункциональное тестирование фокусируется на производительности, безопасности и удобстве использования.
* Производительность: Оценка времени отклика бота на запросы пользователей в условиях повышенной нагрузки.
* Безопасность: Проверка обработки данных пользователей, правильности хранения конфиденциальной информации (например, номеров телефонов).
* Юзабилити: Тестирование интерфейса и логики взаимодействия, чтобы убедиться, что пользователи могут легко использовать бота.

1.3. Тестирование сценариев

* Тестирование сценариев включает в себя проверку предопределенных сценариев взаимодействия пользователя с ботом.
* Позитивные сценарии: Проверка всех ожидаемых действий, например, успешная регистрация, оформление заказа и оплата.
* Негативные сценарии: Тестирование неправильно введенных данных, несуществующих команд и других возможных ошибок, чтобы убедиться, что бот обрабатывает их корректно.

2. Автоматизированное тестирование

Автоматизированное тестирование позволяет значительно ускорить процесс проверки кода и его эффективности.

API-тестирование: бот взаимодействует с веб-сервисами для обработки оплаты или получения меню, протестированы API с помощью инструментов, таких как Postman или Insomnia.

3. Виды тестов

В ходе тестирования чат-бота выполнены следующие виды тестов:

* Модульные тесты: Проверка отдельных функциональных блоков бота, чтобы убедиться в их корректности.
* Интеграционные тесты: Тестирование взаимодействия между различными компонентами системы (например, приложением и базой данных).
* Системные тесты: Оценка всего чат-бота как единого целого, чтобы удостовериться, что он работает в соответствии с требованиями.
* Регрессионные тесты: Проверка функциональности после внесения изменений в код, чтобы убедиться, что ранее работающие функции по-прежнему функционируют корректно.

4. Процесс отладки

Отладка — это процесс обнаружения и исправления ошибок в программном обеспечении.

4.1. Использование логирования

* Логи предоставляют информацию о том, что происходит в боте в процессе его работы, и позволяют разработчикам выявлять ошибки.
* Настройка логирования: Необходимо включить логирование всех действий бота, включая входящие сообщения, обработанные команды и ошибки.

4.2. Интерактивная отладка

Использование отладчиков позволяет в реальном времени просматривать состояние программы.

* Отладчики: Инструменты, такие как pdb (Python Debugger), могут использоваться для пошагового выполнения кода и отслеживания переменных в момент возникновения ошибки.
* Тестовые фреймворки: Многие тестовые фреймворки также имеют встроенные средства для отладки.

5. Подготовка к тестированию

* Создан план тестирования, включающий все ключевые сценарии и методы, которые будут использоваться.
* Подготовлены тестовые данные, включая как валидные, так и невалидные варианты данных, с которыми будет работать бот.

# **ЗАКЛЮЧЕНИЕ**

В результате выполнения дипломной работы был успешно разработан чат-бот доставки еды, который представляет собой современное решение для автоматизации процесса заказа и доставки пищи. Проект включал в себя полный цикл разработки, начиная с анализа требований и проектирования архитектуры и заканчивая реализацией функционала и проведением тестирования. Бот был реализован с использованием Python и библиотеки aiogram.

Разработанный чат-бот обладает рядом ключевых функциональных возможностей, включая:

Регистрация и аутентификация пользователей: Пользователи могут легко зарегистрироваться и войти в систему, что позволяет им сохранять свои данные и историю заказов.

Интерактивный процесс оформления заказов: Бот предоставляет пользователям актуальное меню, позволяет легко выбирать блюда и указывать количество порций, а также предлагает варианты доставки и самовывоза.

Обработка платежей: Реализованы различные способы оплаты, включая наличные и безналичные расчеты, что дает пользователям свободу выбора.

Поддержка и взаимодействие с пользователями: Чат-бот включает функционал для сбора обратной связи и комментариев, что помогает улучшать качество услуг.

Высокий уровень безопасности: Обеспечена защита данных пользователей и конфиденциальная обработка платежной информации.

В процессе разработки был уделен особый внимание тестированию и отладке системы, что позволило избежать критических ошибок и обеспечить надежность работы чат-бота. Использование различных методов тестирования позволило выявить и исправить потенциальные уязвимости и улучшить пользовательский интерфейс.

Кроме того, реализация проекта изучила и применяла лучшие практики разработки чат-ботов, такие как применение архитектуры с разделением на модули, использование состояний для управления логикой взаимодействия с пользователем и использование фреймворков для автоматизации тестирования.

Перспективы развития

Разработка чат-бота доставки еды открывает возможности для будущих улучшений и расширения функционала. В дальнейшем можно рассмотреть внедрение следующих возможностей:

Расширение ассортимента меню: Добавление новых блюд и категорий продуктов для увеличения выбора для пользователей.

Интеграция с системами лояльности: Реализация программ лояльности и акций для постоянных клиентов.

В заключение, разработка чат-бота доставки еды является успешным примером применения современных технологий для улучшения пользовательского опыта и оптимизации процессов в сфере общественного питания. Этот проект может служить основой для дальнейших исследований и разработок в области автоматизации сервисов и улучшения взаимодействия между бизнесом и клиентами.

**СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

1. Официальная документация Django: docs.djangoproject.com
2. Официальная документация PostgreSQL.: postgresql.org/docs
3. Иванов, В. И. Python для разработчиков / В. И. Иванов. — Москва: Диалектика, 2020. — 350 с.
4. Документация WireGuard. https://www.wireguard.com. [Дата обращения: 19.12.2024].
5. Python Software Foundation. Python 3 Documentation. https://docs.python.org/3/. [Дата обращения: 19.12.2024].
6. Морозов, И. В. Основы работы с базами данных MySQL / И. В. Морозов. — Москва: Бином, 2020. — 220 с.

Приложение 1. **Скриншоты интерфейса**

Включены визуальные примеры пользовательского интерфейса для наглядного представления функциональности

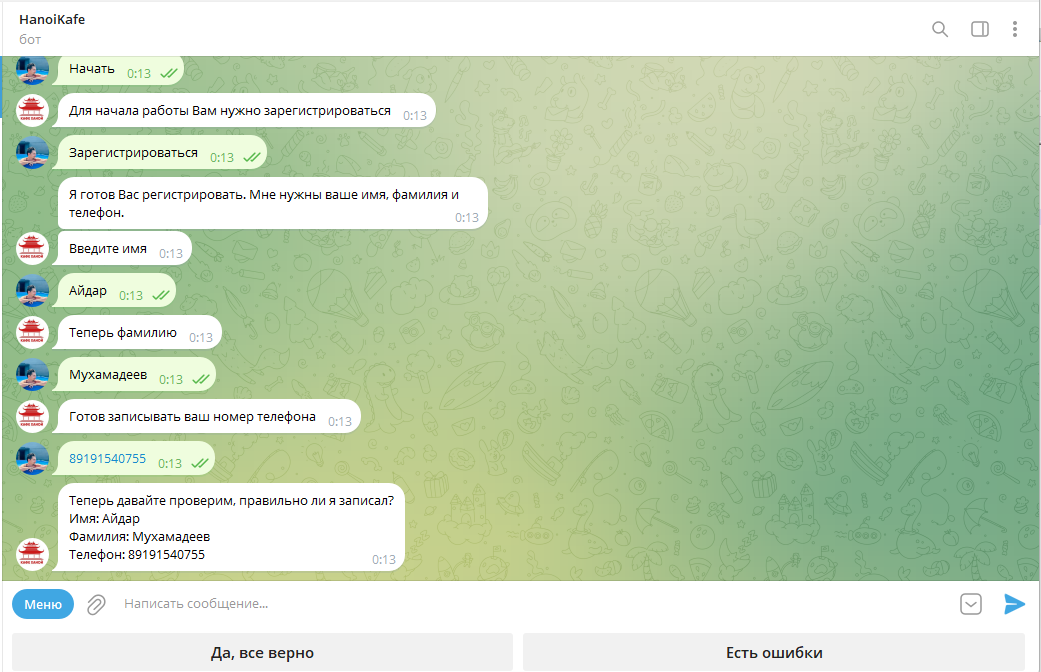


Рис.2 Меню регистрации пользователей Бота

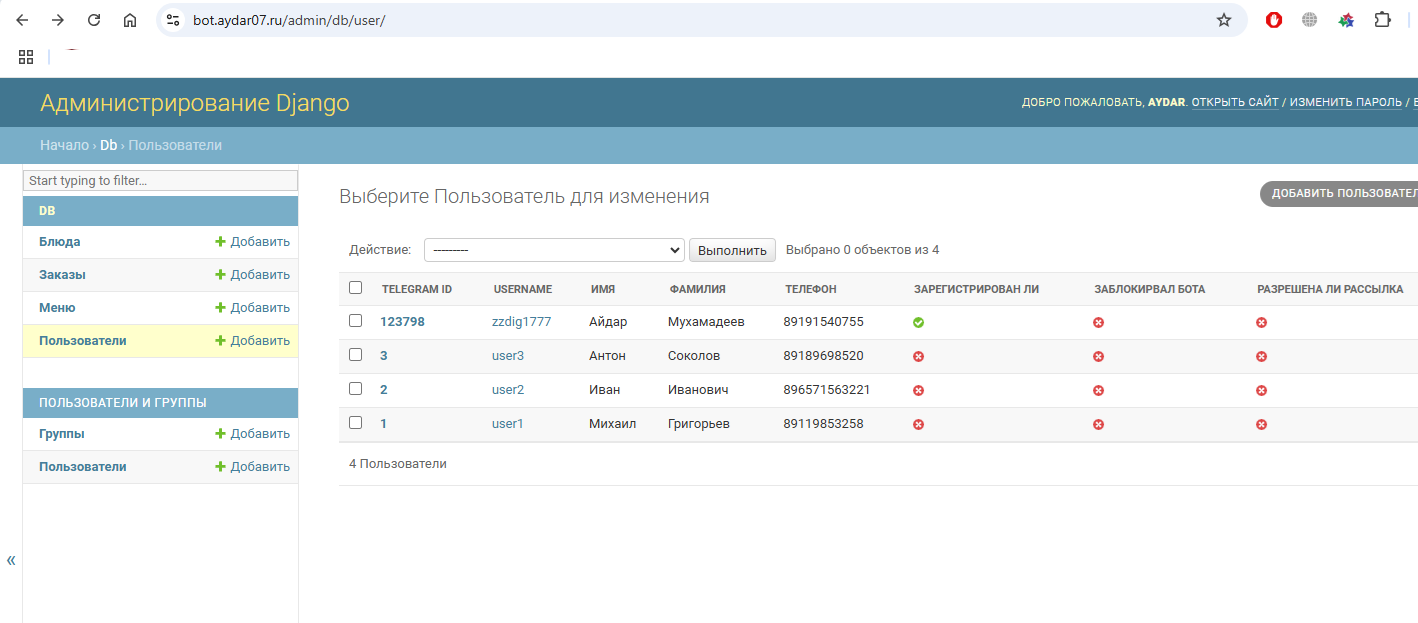


Рис.3 Меню регистрации пользователей

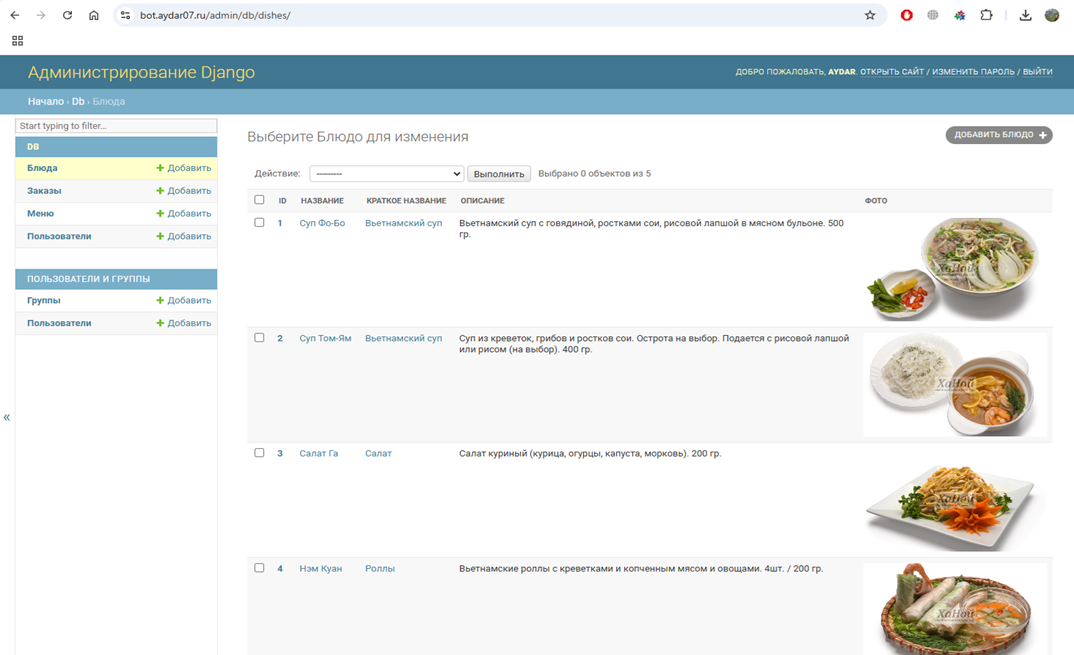
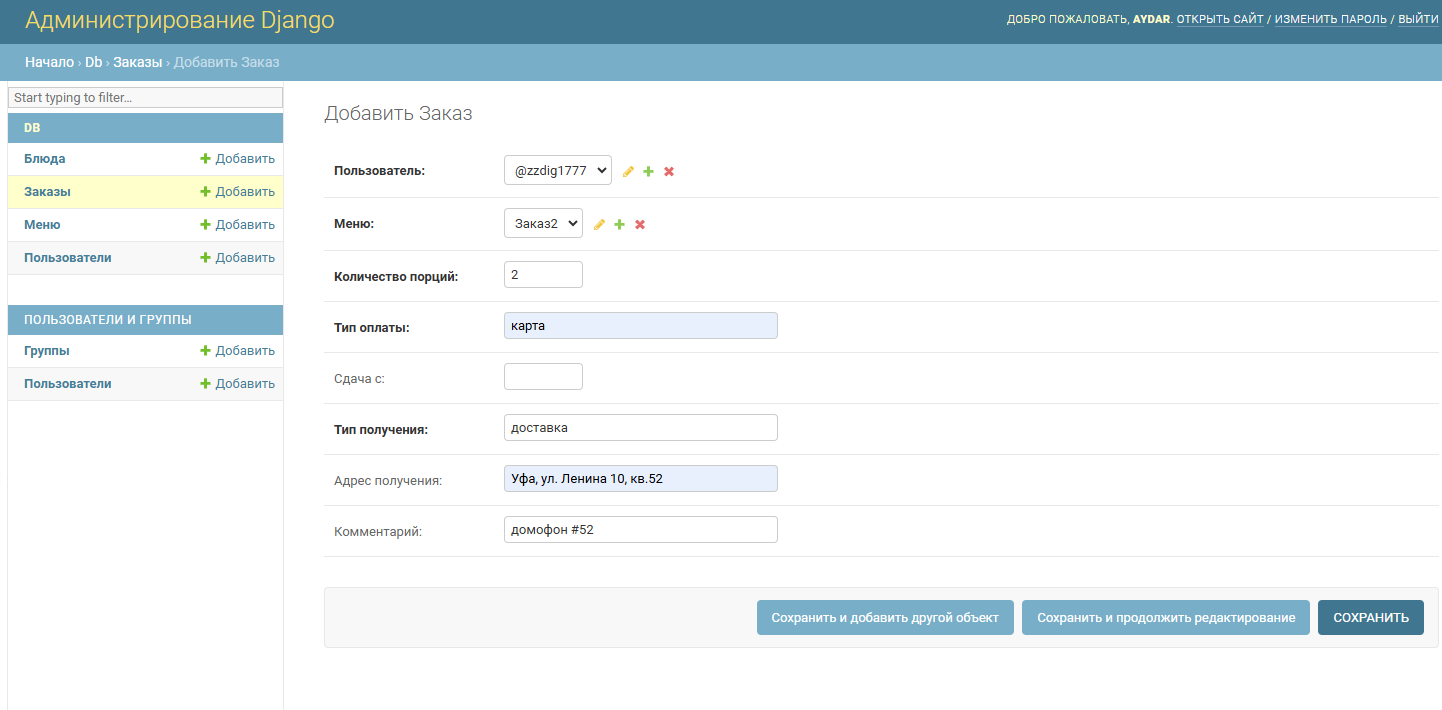
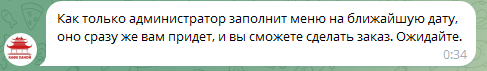


Рис.4 Каталог блюд в панели администратора Django



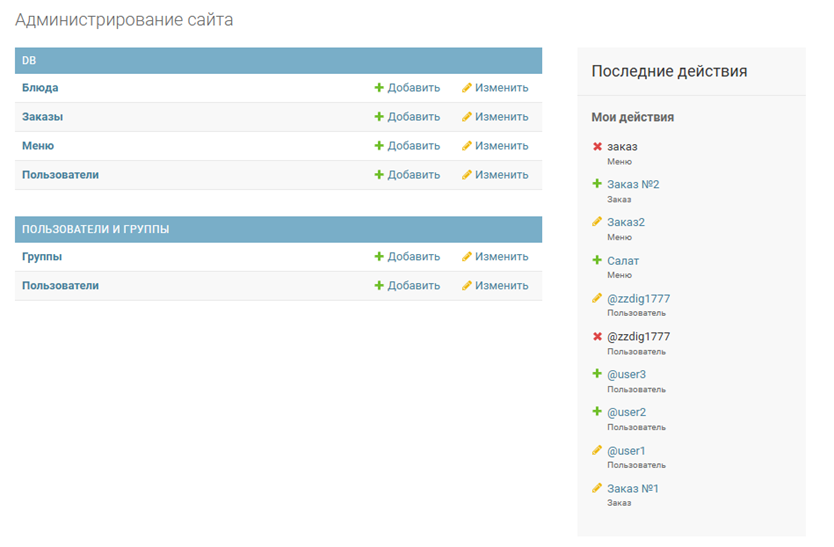
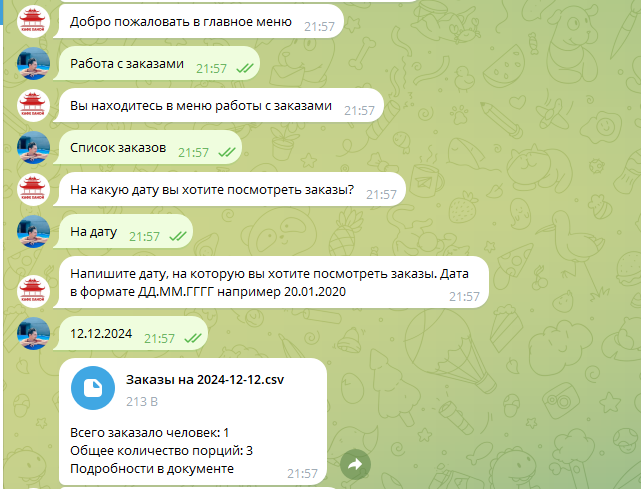


Рис.4 Добавление заказа

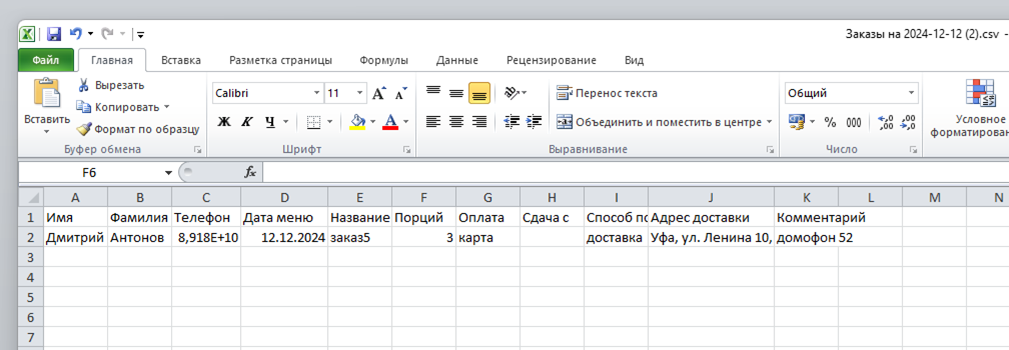
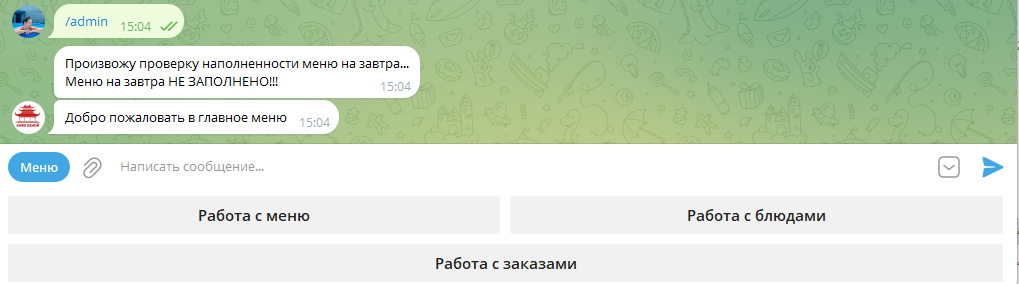


Рис.5 Обработка заказа



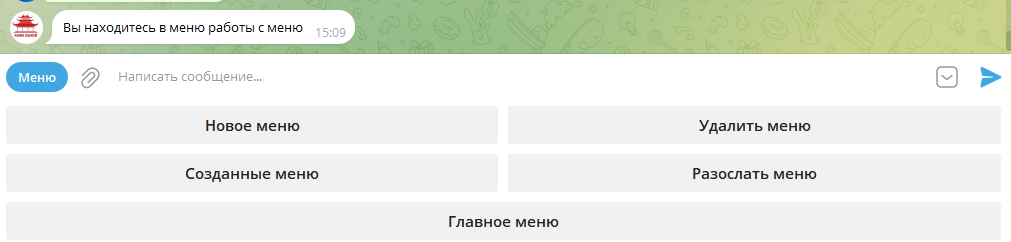
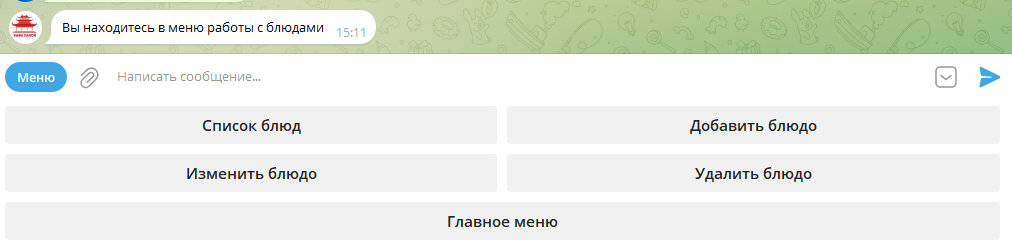


Рис.6 Панель администратора чат-бота (меню)



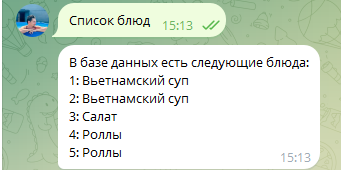
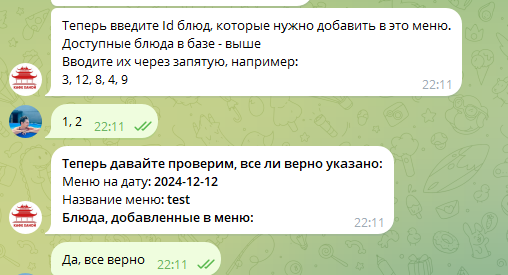
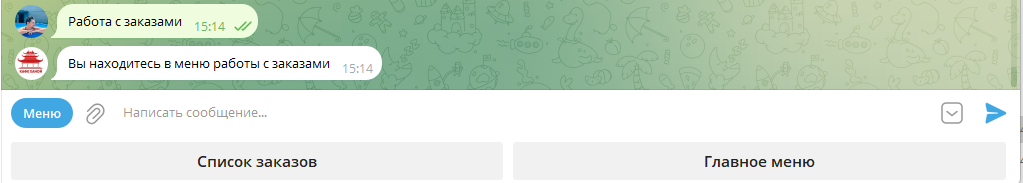
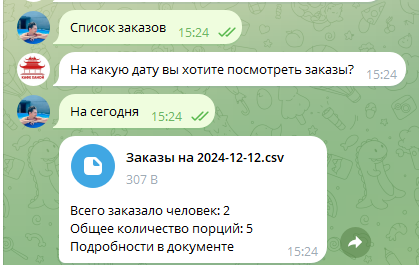
 

Рис.7 Панель администратора чат-бота (блюда)





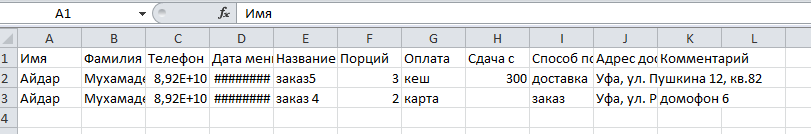


Рис.8 Панель администратора чат-бота (заказы)

**Приложение 2. Код веб-сайта (основные скрипты)**

|  |
| --- |
| **/env.bot**  TELEGRAM\_TOKEN=\*\*\*\*\*\*\*\*\*\*\*  USERS\_URL=http://web:8000/api/users/  DISHES\_URL=http:/web:8000/api/dishes/  MENUS\_URL=http:/web:8000/api/menus/  ORDERS\_URL=http:/web:8000/api/orders/  ADMIN\_IDS=\*\*\*\*\*\*\*\*\*\*\*  IS\_REDIS\_STORAGE=True  IS\_WEBHOOK=True  REDIS\_DSN=redis://redis:6379/0  BASE\_URL=https://bot.aydar07.ru  WEB\_SERVER\_HOST = "0.0.0.0"  WEB\_SERVER\_PORT = 7771  WEBHOOK\_PATH = "/webhook/main/" |

|  |
| --- |
| **handlers/admin\_handlers.py**  # ==================админский доступ========================= from datetime import date, timedelta  from aiogram import types from aiogram.dispatcher.fsm.context import FSMContext  from bot\_app.bot\_create import bot from bot\_app.data\_exchanger import get\_user\_info, list\_menus from bot\_app.keyboards.admin\_kb import admin\_main\_menu\_kb from bot\_app.phrases.admin\_phrases import ADM\_ENTER\_ERR, ADM\_MENU from bot\_app.states import AdminState   async def enter\_admin\_section(message: types.Message, state: FSMContext):  *"""Вход в раздел админа."""* user\_id = message.from\_user.id  response = await get\_user\_info(user\_id)  if not response['json']['is\_admin']:  await bot.send\_message(text=ADM\_ENTER\_ERR,  chat\_id=message.from\_user.id)  else:  await state.set\_state(AdminState.wait\_admin\_check\_today\_menu)  return await admin\_check\_today\_menu(message, state)   async def admin\_check\_today\_menu(message: types.Message, state: FSMContext):  *"""Проверка на заполненность меню на завтра. Происходит каждый раз при  входе в раздел админа."""* menus = await list\_menus()  tomorrow = date.today() + timedelta(days=1)  text = 'Произвожу проверку наполненности меню на завтра...'  if not any(menu['date\_of\_menu'] == tomorrow for menu in menus):  text += '\nМеню на завтра НЕ ЗАПОЛНЕНО!!!'  else:  text += '\nВы уже наполнили меню на завтра, все ОК'  await bot.send\_message(text=text,  chat\_id=message.from\_user.id)  await state.set\_state(AdminState.wait\_admin\_main\_menu)  return await admin\_main\_menu(message, state)   async def admin\_main\_menu(message: types.Message, state: FSMContext):  *"""Главное меню администратора."""* await bot.send\_message(text=ADM\_MENU,  chat\_id=message.from\_user.id,  reply\_markup=admin\_main\_menu\_kb())  await state.set\_state(AdminState.wait\_admin\_choose\_main\_menu) |

|  |
| --- |
| **handlers /dishes\_handlers.py**  # ==================работа с блюдами================== from aiogram import types from aiogram.dispatcher.fsm.context import FSMContext from aiogram.exceptions import TelegramBadRequest  from bot\_app.bot\_create import BASE\_URL from bot\_app.bot\_create import bot from bot\_app.data\_exchanger import (list\_dishes, add\_dish, get\_dish,  edit\_dish, remove\_dish) from bot\_app.handlers.admin.admin\_handlers import admin\_main\_menu from bot\_app.keyboards.admin\_kb import (admin\_work\_with\_dishes\_kb,  admin\_add\_dish\_noimage\_kb,  admin\_add\_dish\_check\_kb,  admin\_start\_repair\_dish\_kb,  admin\_remove\_dish\_choose\_kb) from bot\_app.keyboards.common\_kb import remove\_kb from bot\_app.phrases.admin\_phrases import (DISH\_MENU\_TEXT, ADM\_CHOOSE\_DISH,  ADM\_CHOOSE\_DISH\_2, LIST\_DISH,  ADM\_ADD\_TITLE, ADM\_ADD\_DESCR,  ADM\_ADD\_PHOTO, ADD\_TO\_DB,  DISH\_REPAIR, DISH\_REPAIR\_2,  DISHES\_TO\_REPAIR,  CHECK\_ID\_ERR, ID\_DISH\_404,  ID\_DISH\_OTHER\_ERR,  START\_REPAIR\_DISH,  START\_REPAIR\_DISH\_2, DISH\_EDIT\_DATA,  DISH\_EDIT\_DATA\_2, DISH\_EDIT\_DATA\_3,  DISH\_EDIT\_DATA\_4, DISH\_EDIT\_SUCCESS,  DISH\_EDIT\_ERR, DISH\_RM\_ENTER\_ID,  DISH\_RM\_ACCEPT, DISH\_RM\_CONFIRM,  DISH\_RM\_CONFIRM\_2,  DISH\_RM\_CONFIRM\_3,  DISH\_RM\_CANCEL) from bot\_app.states import AdminState   async def admin\_work\_with\_dishes(message: types.Message, state: FSMContext):  *"""Меню работы с блюдами."""* await bot.send\_message(text=DISH\_MENU\_TEXT,  chat\_id=message.from\_user.id,  reply\_markup=admin\_work\_with\_dishes\_kb())  await state.set\_state(AdminState.wait\_admin\_choose\_dishes)   async def admin\_choose\_dishes(message: types.Message, state: FSMContext):  *"""Обработка кнопок из меню работы с блюдами."""* if message.text == 'Список блюд':  text = await admin\_list\_dishes()  await bot.send\_message(text=text,  chat\_id=message.from\_user.id)  await state.set\_state(AdminState.wait\_admin\_work\_with\_dishes)  return await admin\_work\_with\_dishes(message, state)  elif message.text == 'Добавить блюдо':  await bot.send\_message(text=ADM\_CHOOSE\_DISH,  chat\_id=message.from\_user.id)  await bot.send\_message(text=ADM\_CHOOSE\_DISH\_2,  chat\_id=message.from\_user.id,  reply\_markup=remove\_kb())  await state.set\_state(AdminState.wait\_admin\_add\_dish\_title)  elif message.text == 'Изменить блюдо':  await state.set\_state(AdminState.wait\_admin\_repair\_dish)  return await admin\_repair\_dish(message, state)  elif message.text == 'Удалить блюдо':  await state.set\_state(AdminState.wait\_admin\_remove\_dish)  return await admin\_remove\_dish(message, state)  elif message.text == 'Главное меню':  await state.set\_state(AdminState.wait\_admin\_main\_menu)  return await admin\_main\_menu(message, state)   # Список доступных блюд async def admin\_list\_dishes():  *"""Представление списка доступных блюд в базе данных"""* response = await list\_dishes()  text = LIST\_DISH  for item in response['json']:  text += f'{item["id"]}: {item["shortname"]}\n'  return text   # Добавление блюда async def admin\_add\_dish\_title(message: types.Message, state: FSMContext):  *"""Добавление нового блюда. Краткое название блюда."""* await state.update\_data(title=message.text)  await bot.send\_message(text=ADM\_ADD\_TITLE,  chat\_id=message.from\_user.id)  await state.set\_state(AdminState.wait\_admin\_add\_dish\_shortname)   async def admin\_add\_dish\_shortname(message: types.Message, state: FSMContext):  *"""Добавление нового блюда. Описание блюда."""* await state.update\_data(shortname=message.text)  await bot.send\_message(text=ADM\_ADD\_DESCR,  chat\_id=message.from\_user.id)  await state.set\_state(AdminState.wait\_admin\_add\_dish\_descr)   async def admin\_add\_dish\_descr(message: types.Message, state: FSMContext):  *"""Добавление нового блюда. Фотография блюда, либо без фотографии."""* await state.update\_data(descr=message.text)  await bot.send\_message(text=ADM\_ADD\_PHOTO,  chat\_id=message.from\_user.id,  reply\_markup=admin\_add\_dish\_noimage\_kb())  await state.set\_state(AdminState.wait\_admin\_add\_dish\_photo)   async def admin\_add\_dish\_photo(message: types.Message, state: FSMContext):  *"""Добавление нового блюда. Обработка информации, фотографии и вывод  пользователю на подтверждение."""* data = await state.get\_data()  text = ''  if message.photo:  mes\_img = message.photo[-1]  file\_id = mes\_img.file\_id  file = await bot.get\_file(file\_id)  filename = mes\_img.file\_id  await state.update\_data(image=file.file\_path,  filename=filename,  image\_id=file\_id)  text = (f'Теперь проверим как это выглядит:\n'  f'Название: {data["title"]}\n'  f'Короткое название: {data["shortname"]}\n'  f'Описание: {data["descr"]}\n'  f'Фото: на которое отвечаю')  if message.text:  await state.update\_data(image=None,  filename=None,  image\_id=None)  text = (f'Теперь проверим как это выглядит:\n'  f'Название: {data["title"]}\n'  f'Короткое название: {data["shortname"]}\n'  f'Описание: {data["descr"]}\n'  f'Фото: Без фото')  await message.reply(text=text,  reply\_markup=admin\_add\_dish\_check\_kb())  await state.set\_state(AdminState.wait\_admin\_confirm\_add\_dish\_data)   async def admin\_confirm\_add\_dish\_data(message: types.Message,  state: FSMContext):  *"""Сохранение блюда в базу данных в случае подтверждения, возврат в  начало добавления блюда в случае отказа."""* data = await state.get\_data()  if message.text == 'Да, все верно':  image = None  if data["image"]:  image\_path = data["image"]  image = (await bot.download\_file(image\_path)).read()  dish\_data = {  'title': data["title"],  'shortname': data["shortname"],  'description': data["descr"],  'image': image,  'image\_id': data["image\_id"],  'filename': data["filename"],  }  await add\_dish(dish\_data)  await bot.send\_message(text=ADD\_TO\_DB,  chat\_id=message.from\_user.id)  await state.set\_state(AdminState.wait\_admin\_work\_with\_dishes)  return await admin\_work\_with\_dishes(message, state)   elif message.text == 'Нужно исправить':  await bot.send\_message(text=DISH\_REPAIR,  chat\_id=message.from\_user.id)  await bot.send\_message(text=DISH\_REPAIR\_2,  chat\_id=message.from\_user.id,  reply\_markup=remove\_kb())  await state.set\_state(AdminState.wait\_admin\_add\_dish\_title)   # исправление блюда async def admin\_repair\_dish(message: types.Message, state: FSMContext):  *"""Исправление блюда. Ввод ID блюда из базы."""* dishes = await admin\_list\_dishes()  text = (f'{dishes}\n'  f'{DISHES\_TO\_REPAIR}')  await bot.send\_message(text=text,  chat\_id=message.from\_user.id,  reply\_markup=remove\_kb())  await state.set\_state(AdminState.wait\_admin\_start\_repair\_dish)   async def admin\_check\_id\_dish\_value(message: types.Message):  *"""Проверка введенного ID блюда на валидность"""* dish\_id = message.text  response = await get\_dish(dish\_id)  try:  int(dish\_id)  except ValueError:  await bot.send\_message(text=CHECK\_ID\_ERR,  chat\_id=message.from\_user.id)  return response['json'], False  if response['status'] == 404:  await bot.send\_message(text=ID\_DISH\_404,  chat\_id=message.from\_user.id)  return response['json'], False  elif response['status'] == 200:  return response['json'], True  else:  await bot.send\_message(text=ID\_DISH\_OTHER\_ERR,  chat\_id=message.from\_user.id)  return response['json'], False   async def admin\_get\_formalized\_dish(message: types.Message, dish\_instance):  *"""Стилизованный вывод блюда."""* dish = dish\_instance  dish\_title = dish['title']  dish\_shortname = dish['shortname']  dish\_descr = dish['description']  dish\_image = dish['image']  dish\_filename = f"{BASE\_URL}/{dish['filename']}"  dish\_image\_id = dish['image\_id']  formalized\_dish = (f'<b>Название</b>\n{dish\_title}\n'  f'<b>Краткое название</b>\n{dish\_shortname}\n'  f'<b>Описание</b>\n{dish\_descr}')  if dish\_image is None:  formalized\_dish += '\n<b>Фото</b>\nБез фото'  await bot.send\_message(text=formalized\_dish,  chat\_id=message.from\_user.id)  else:  if dish\_image\_id is None:  await bot.send\_photo(message.from\_user.id, dish\_filename,  formalized\_dish)  else:  try:  await bot.send\_photo(message.from\_user.id, dish\_image\_id,  formalized\_dish)  except TelegramBadRequest:  await bot.send\_photo(message.from\_user.id,  dish\_filename,  formalized\_dish)   async def admin\_start\_repair\_dish(message: types.Message, state: FSMContext):  *"""Вывод стилизованного меню админу и выбор параметра блюда, который  нужно изменить."""* dish\_instance, is\_correct\_digit = await admin\_check\_id\_dish\_value(  message)  if not is\_correct\_digit:  await state.set\_state(AdminState.wait\_admin\_repair\_dish)  return await admin\_repair\_dish(message, state)  await state.update\_data(dish\_instance=dish\_instance)  await bot.send\_message(text=START\_REPAIR\_DISH,  chat\_id=message.from\_user.id)  await admin\_get\_formalized\_dish(message, dish\_instance)  await bot.send\_message(text=START\_REPAIR\_DISH\_2,  chat\_id=message.from\_user.id,  reply\_markup=admin\_start\_repair\_dish\_kb())  await state.set\_state(AdminState.wait\_admin\_repair\_dish\_choose\_field)   async def admin\_repair\_dish\_choose\_field(message: types.Message,  state: FSMContext):  *"""Запрос нового значения для выбранного параметра блюда."""* db\_key\_to\_change = None  if message.text == 'Название':  db\_key\_to\_change = 'title'  await bot.send\_message(text=DISH\_EDIT\_DATA,  chat\_id=message.from\_user.id,  reply\_markup=remove\_kb())  elif message.text == 'Краткое название':  db\_key\_to\_change = 'shortname'  await bot.send\_message(text=DISH\_EDIT\_DATA\_2,  chat\_id=message.from\_user.id,  reply\_markup=remove\_kb())  elif message.text == 'Описание':  db\_key\_to\_change = 'description'  await bot.send\_message(text=DISH\_EDIT\_DATA\_3,  chat\_id=message.from\_user.id,  reply\_markup=remove\_kb())  elif message.text == 'Фотография':  db\_key\_to\_change = 'image'  await bot.send\_message(text=DISH\_EDIT\_DATA\_4,  chat\_id=message.from\_user.id,  reply\_markup=admin\_add\_dish\_noimage\_kb())  await state.update\_data(db\_key\_to\_change=db\_key\_to\_change)  await state.set\_state(AdminState.wait\_admin\_repair\_dish\_fields)   async def admin\_repair\_dish\_fields(message: types.Message, state: FSMContext):  *"""Внесение изменений в указанный параметр блюда, его сохранение в базе  данных."""* new\_dish\_data = {}  data = await state.get\_data()  dish\_id = data['dish\_instance']['id']  key = data['db\_key\_to\_change']  if message.photo:  mes\_img = message.photo[-1]  file\_id = mes\_img.file\_id  file = await bot.get\_file(file\_id)  filename = mes\_img.file\_id  image = (await bot.download\_file(file.file\_path)).read()  new\_dish\_data['image'] = image  new\_dish\_data['filename'] = filename  new\_dish\_data['image\_id'] = file\_id  if message.text:  if message.text == 'Без фото':  new\_dish\_data['image'] = None  new\_dish\_data['filename'] = None  new\_dish\_data['image\_id'] = None  else:  new\_dish\_data[key] = message.text  response = await edit\_dish(new\_dish\_data, dish\_id)  if response['status'] == 200:  await bot.send\_message(text=DISH\_EDIT\_SUCCESS,  chat\_id=message.from\_user.id,  reply\_markup=admin\_work\_with\_dishes\_kb())  await state.set\_state(AdminState.wait\_admin\_work\_with\_dishes)  return await admin\_work\_with\_dishes(message, state)  else:  await bot.send\_message(text=DISH\_EDIT\_ERR,  chat\_id=message.from\_user.id,  reply\_markup=remove\_kb())  await state.set\_state(AdminState.wait\_admin\_work\_with\_dishes)  return await admin\_work\_with\_dishes(message, state)   # удаление блюда async def admin\_remove\_dish(message: types.Message, state: FSMContext):  *"""Удаление блюда. Запрос ID блюда для удаления."""* await bot.send\_message(text=DISH\_RM\_ENTER\_ID,  chat\_id=message.from\_user.id,  reply\_markup=remove\_kb())  await state.set\_state(AdminState.wait\_admin\_remove\_dish\_choose)   async def admin\_remove\_dish\_choose(message: types.Message, state: FSMContext):  *"""Проверка введенного ID блюда на валидность, вывод админу блюда,  запрос подтверждения удаления этого блюда."""* dish\_instance, is\_correct\_digit = await admin\_check\_id\_dish\_value(  message)  if not is\_correct\_digit:  await state.set\_state(AdminState.wait\_admin\_repair\_dish)  return await admin\_repair\_dish(message, state)  await state.update\_data(dish\_id=message.text)  await admin\_get\_formalized\_dish(message, dish\_instance)  await bot.send\_message(text=DISH\_RM\_ACCEPT,  chat\_id=message.from\_user.id,  reply\_markup=admin\_remove\_dish\_choose\_kb())  await state.set\_state(AdminState.wait\_admin\_remove\_dish\_confirm)   async def admin\_remove\_dish\_confirm(message: types.Message, state: FSMContext):  *"""Обработка подтверждения / отмены для удаления блюда."""* if message.text == 'Да, уверен':  data = await state.get\_data()  dish\_id = data['dish\_id']  response = await remove\_dish(dish\_id)  if response['status'] == 204:  text = DISH\_RM\_CONFIRM  elif response['status'] == 404:  text = DISH\_RM\_CONFIRM\_2.format(response)  else:  text = DISH\_RM\_CONFIRM\_3.format(response)  await bot.send\_message(text=text,  chat\_id=message.from\_user.id,  reply\_markup=admin\_work\_with\_dishes\_kb())  await state.set\_state(AdminState.wait\_admin\_work\_with\_dishes)  return await admin\_work\_with\_dishes(message, state)   elif message.text == 'Отмена!!':  await bot.send\_message(text=DISH\_RM\_CANCEL,  chat\_id=message.from\_user.id,  reply\_markup=admin\_work\_with\_dishes\_kb())  await state.set\_state(AdminState.wait\_admin\_work\_with\_dishes)  return await admin\_work\_with\_dishes(message, state)   async def client\_get\_formalized\_dish(dish\_instance, client\_id):  *"""Красивое представление меню для пользователя."""* dish = dish\_instance  dish\_title = dish['title']  dish\_descr = dish['description']  dish\_image = dish['image']  dish\_filename = f"{BASE\_URL}/{dish['filename']}"  dish\_image\_id = dish['image\_id']  formalized\_dish = (f'<b>Блюдо:</b>\n{dish\_title}\n'  f'<b>Описание: </b>\n{dish\_descr}')  if dish\_image is None:  formalized\_dish += '\n<b>Фото</b>\nБез фото'  await bot.send\_message(text=formalized\_dish,  chat\_id=client\_id)  else:  if dish\_image\_id is None:  await bot.send\_photo(client\_id, dish\_filename,  formalized\_dish)  else:  try:  await bot.send\_photo(client\_id, dish\_image\_id,  formalized\_dish)  except TelegramBadRequest:  print(dish\_filename)  await bot.send\_photo(client\_id,  dish\_filename,  formalized\_dish) |

|  |
| --- |
| **handlers /registration\_handlers.py**  import json  from aiogram import types from aiogram.dispatcher.fsm.context import FSMContext  from bot\_app.bot\_create import bot from bot\_app.data\_exchanger import get\_user\_info, user\_registration from bot\_app.handlers.user\_handlers import client\_msg\_before\_start from bot\_app.handlers.utils import (validate\_phone, get\_data\_to\_validate,  validate\_name) from bot\_app.keyboards.common\_kb import (start\_using, reg\_kb, remove\_kb,  ready\_repair\_reg\_kb,  confirm\_reg\_kb) from bot\_app.phrases.user\_phrases import (CHECK\_REG\_1, CHECK\_REG\_2,  START\_REG\_1,  START\_REG\_2, INPUT\_NAME\_1,  INPUT\_NAME\_2, INPUT\_LASTNAME\_1,  INPUT\_LASTNAME\_2, INPUT\_PHONE\_1,  INPUT\_PHONE\_2, CONFIRM\_REG\_1,  CONFIRM\_REG\_ERR, CONFIRM\_REG\_2) from bot\_app.states import UserState, RegState   async def check\_registration(message: types.Message, state: FSMContext):  *"""Проверка на регистрацию."""* user\_id = message.from\_user.id  response = await get\_user\_info(user\_id)  if response['json']['is\_register'] and response['json']['is\_allow\_mail']:  await state.set\_state(UserState.wait\_to\_waiting\_order\_info)  return await client\_msg\_before\_start(message)  elif (response['json']['is\_register']  and not response['json']['is\_allow\_mail']):  await bot.send\_message(text=CHECK\_REG\_1,  chat\_id=message.from\_user.id,  reply\_markup=start\_using())  await state.set\_state(UserState.wait\_to\_enter\_client\_section)  else:  await bot.send\_message(text=CHECK\_REG\_2,  chat\_id=message.from\_user.id,  reply\_markup=reg\_kb())  await state.set\_state(RegState.wait\_start\_registration)   async def start\_registration(message: types.Message, state: FSMContext):  *"""Начало регистрации."""* await bot.send\_message(text=START\_REG\_1,  chat\_id=message.from\_user.id,  reply\_markup=remove\_kb())  await bot.send\_message(text=START\_REG\_2,  chat\_id=message.from\_user.id)  await state.set\_state(RegState.wait\_input\_firstname)   async def input\_firstname(message: types.Message, state: FSMContext):  *"""Ввод имени."""* name = validate\_name(message)  if not name:  await bot.send\_message(text=INPUT\_NAME\_1,  chat\_id=message.from\_user.id,  reply\_markup=remove\_kb())  return await get\_data\_to\_validate(state)  await state.update\_data(input\_firstname=name)  await bot.send\_message(text=INPUT\_NAME\_2,  chat\_id=message.from\_user.id,  reply\_markup=remove\_kb())  await state.set\_state(RegState.wait\_input\_lastname)   async def input\_lastname(message: types.Message, state: FSMContext):  *"""Ввод фамилии."""* surname = validate\_name(message)  if not surname:  await bot.send\_message(text=INPUT\_LASTNAME\_1,  chat\_id=message.from\_user.id,  reply\_markup=remove\_kb())  return await get\_data\_to\_validate(state)  await state.update\_data(input\_lastname=surname)  await bot.send\_message(text=INPUT\_LASTNAME\_2,  chat\_id=message.from\_user.id,  reply\_markup=remove\_kb())  await state.set\_state(RegState.wait\_input\_phone)   async def input\_phone(message: types.Message, state: FSMContext):  *"""Ввод номера телефона."""* phone = validate\_phone(message)  if not phone:  await bot.send\_message(text=INPUT\_PHONE\_1,  chat\_id=message.from\_user.id,  reply\_markup=remove\_kb())  return await get\_data\_to\_validate(state)  await state.update\_data(input\_phone=phone)  data = await state.get\_data()  text = (f'{INPUT\_PHONE\_2}\n'  f'Имя: {data["input\_firstname"]}\n'  f'Фамилия: {data["input\_lastname"]}\n'  f'Телефон: {data["input\_phone"]}\n')  await bot.send\_message(text=text,  chat\_id=message.from\_user.id,  reply\_markup=confirm\_reg\_kb())  await state.set\_state(RegState.wait\_confirm\_registration)   async def confirm\_registration(message: types.Message, state: FSMContext):  *"""Подтверждение регистрации."""* if message.text == 'Да, все верно':  data = await state.get\_data()  user\_id = str(message.from\_user.id)  user\_data = {  'input\_firstname': data["input\_firstname"],  'input\_lastname': data["input\_lastname"],  'input\_phone': data["input\_phone"],  'is\_register': True,  }  json\_data = json.loads(json.dumps(user\_data))  response = await user\_registration(user\_id, json\_data)  data.clear()  if response['status'] == 200:  await bot.send\_message(text=CONFIRM\_REG\_1,  chat\_id=message.from\_user.id,  reply\_markup=start\_using())  await state.set\_state(UserState.wait\_to\_enter\_client\_section)  else:  await bot.send\_message(text=CONFIRM\_REG\_ERR,  chat\_id=message.from\_user.id)  await bot.send\_message(text=str(json\_data),  chat\_id=message.from\_user.id)  if message.text == 'Есть ошибки':  await bot.send\_message(text=CONFIRM\_REG\_2,  chat\_id=message.from\_user.id,  reply\_markup=ready\_repair\_reg\_kb())  await state.set\_state(RegState.wait\_start\_registration) |

|  |
| --- |
| **handlers /utils.py**  import csv import re from datetime import datetime  from aiogram import types from aiogram.dispatcher.fsm.context import FSMContext from aiogram.types import FSInputFile  from bot\_app.bot\_create import bot from bot\_app.keyboards.common\_kb import remove\_kb from bot\_app.phrases.common\_phrases import VALIDATE\_DATE\_TEXT   async def get\_data\_to\_validate(state: FSMContext):  *"""Запрос данных для валидации"""* def validate\_phone(message: types.Message):  *"""Проверка введенного номера телефона"""* phone = message.text  pattern = re.compile(  r'^((8|\+7)[\- ]?)?(\(?\d{3}\)?[\- ]?)?[\d\- ]{7,10}$')  if not re.match(pattern, phone):  return False  else:  return phone   def validate\_name(message: types.Message):  *"""Проверка введенного имени или фамилии"""* name = message.text  pattern = re.compile(  r'^[-a-zA-Zа-яА-Яё]+$')  if not re.match(pattern, name):  return False  else:  return name   async def validate\_date(message: types.Message, my\_date):  *"""Проверка на валидность введенной даты. Принимается дата в формате  ДД.ММ.ГГГГ ."""* pattern = re.compile('(^(0[1-9]|[12][0-9]|3[01])[.](0[1-9]|1[012])[.]('  '19|20)[0-9]{2}$)')  if not re.match(pattern, my\_date):  await bot.send\_message(text=VALIDATE\_DATE\_TEXT,  chat\_id=message.from\_user.id,  reply\_markup=remove\_kb())  formalized\_date = None  return formalized\_date  else:  formalized\_date = datetime.strptime(my\_date, "%d.%m.%Y").date()  return formalized\_date   def create\_csv\_file(orders):   fieldnames = ['Имя', 'Фамилия', 'Телефон', 'Дата меню',  'Название меню', 'Порций', 'Оплата', 'Сдача с',  'Способ получения', 'Адрес доставки', 'Комментарий']  cnt\_users = 0  cnt\_servings = 0  with open('orders.csv', mode='w', encoding='windows-1251',  newline='') as f:  writer = csv.writer(f, delimiter=';')  writer.writerow(fieldnames)  for order in orders['json']:  csvdata = [  order['user']['input\_firstname'],  order['user']['input\_lastname'],  order['user']['input\_phone'],  order['menu']['date\_of\_menu'],  order['menu']['title'],  order['num\_of\_servings'],  order['payment\_type'],  order['cash\_change'],  order['delivery'],  order['delivery\_address'],  order['comment']  ]  writer.writerow(csvdata)  cnt\_users += 1  cnt\_servings += int(order['num\_of\_servings'])  caption = (f'Всего заказало человек: {cnt\_users} \n'  f'Общее количество порций: {cnt\_servings}\n'  f'Подробности в документе')  f.close()  file = FSInputFile("orders.csv",  filename=f"Заказы на"  f" {order['menu']['date\_of\_menu']}.csv")  return file, caption   def check\_case\_serving(count):  *"""Подбор правильного окончания для слово "порция"."""* mes1 = 'порция'  mes2 = 'порции'  mes3 = 'порций'  mes\_count = ''  try:  count = int(count)  if count % 10 == 1:  mes\_count = mes1  elif 1 <= count % 10 <= 4:  mes\_count = mes2  elif 5 <= count % 10 <= 9 or count % 10 == 0:  mes\_count = mes3  return mes\_count  except TypeError:  pass |

|  |
| --- |
| **bot\_app/bot\_create.py**  import os  from aiogram import Bot, Dispatcher from aiogram.client.session.aiohttp import AiohttpSession from aiogram.dispatcher.fsm.storage.memory import MemoryStorage from aiogram.dispatcher.fsm.storage.redis import RedisStorage from aiogram.dispatcher.webhook.aiohttp\_server import (SimpleRequestHandler,  setup\_application) from aiohttp import web from dotenv import load\_dotenv  load\_dotenv()  TELEGRAM\_TOKEN = os.getenv('TELEGRAM\_TOKEN') REDIS\_DSN = os.getenv('REDIS\_DSN')  BASE\_URL = os.getenv('BASE\_URL') WEB\_SERVER\_HOST = os.getenv('WEB\_SERVER\_HOST') WEB\_SERVER\_PORT = os.getenv('WEB\_SERVER\_PORT') WEBHOOK\_PATH = os.getenv('WEBHOOK\_PATH') WEBHOOK\_URL = f'{BASE\_URL}{WEBHOOK\_PATH}' IS\_REDIS\_STORAGE = os.getenv('IS\_REDIS\_STORAGE')  session = AiohttpSession() bot\_settings = {  "session": session,  "parse\_mode": "HTML" } bot = Bot(token=TELEGRAM\_TOKEN, \*\*bot\_settings)  if IS\_REDIS\_STORAGE.lower() in ['true', '1', 'yes']:  storage = RedisStorage.from\_url(REDIS\_DSN) else:  storage = MemoryStorage() dp = Dispatcher(storage=storage)   def start\_webhook():  *""" Запускает бота в режиме webhook"""* app = web.Application()  handler = SimpleRequestHandler(dispatcher=dp, bot=bot)  handler.register(app, path=WEBHOOK\_PATH)  setup\_application(app, dp, bot=bot)  web.run\_app(app, host=WEB\_SERVER\_HOST, port=WEB\_SERVER\_PORT) |