**Содержание**

[Введение 2](#_Toc136425488)

[1. Анализ предметной области и выбор технологий для разработки 3](#_Toc136425489)

[1.1. Анализ предметной области 3](#_Toc136425490)

[1.2. Выбор технологий для разработки 4](#_Toc136425491)

[2. Проектирование базы данных 6](#_Toc136425492)

[3. Разработка дизайна приложения 9](#_Toc136425493)

[4. Разработка серверной составляющей веб-приложения 15](#_Toc136425494)

[4.1 Описание маршрутов 16](#_Toc136425495)

[4.2 Описание моделей 20](#_Toc136425496)

[4.3 Описание сериализаторов 23](#_Toc136425497)

[4.4 Функция обработчик 29](#_Toc136425498)

[4.5 Описание представлений 30](#_Toc136425499)

[4.6 Описание миграций 35](#_Toc136425500)

[5. Разработка клиентской составляющей веб-приложения 36](#_Toc136425501)

[6. Реализация программы для поддержки ботов 38](#_Toc136425502)

[7. Руководство пользователя 40](#_Toc136425503)

[7.1. Авторизация 40](#_Toc136425504)

[7.2. Список ботов 41](#_Toc136425505)

[7.3. Конструктор 43](#_Toc136425506)

[7.4. Проверка работы бота 47](#_Toc136425507)

[8. Перспективы 50](#_Toc136425508)

[Заключение 51](#_Toc136425509)

[Список использованных источников 52](#_Toc136425510)

# Введение

Чат-боты становятся все более популярными и широко используемыми в различных сферах, включая бизнес, образование, здравоохранение и многие другие. Использование чат-ботов позволяет автоматизировать процессы, улучшить взаимодействие с клиентами и повысить эффективность работы. С увеличением спроса на чат-боты возрастает и потребность в инструментах, таких как "Конструктор чат-ботов", которые облегчают и ускоряют процесс создания и настройки чат-ботов.

Они помогают имитировать диалог с пользователем путём привязки команд к ключевым фразам. Это помогает избавится от участия человека в информировании пользователей.

Их использование хорошо подходит большим компаниям для автоматизированного ответа на частые вопросы клиентов и сотрудников, отправки файлов и рассылки новостей. Также чат-боты используются для получения различной направленной информации (например, поисковик по базе фильмов).

Создание ботов для мессенджеров является настолько не трудной задачей, что любой человек изучив пару обучающих видео сможет написать его.

На этом этапе проблем не возникает, но увеличивая структуру запросов и создавая новых ботов становится всё труднее их поддерживать, так как даже для программиста возвращение к старому коду создаёт много вопросов и приходится заново изучать то, что было написано ранее. Вывод: такой подход занимает много времени.

Также существуют сайты предлагающие услуги конструктора и поддержки ботов, но конечно это не бесплатно.

В компании было принято решение разработать своё ПО для этих нужд.

## Анализ предметной области и выбор технологий для разработки

Веб-приложение "Конструктор чат-ботов" предназначено для создания и настройки чат-ботов для различных платформ, в данном случае реализуется создание чат-ботов для платформы Telegram. Чат-боты позволяют автоматизировать процесс общения с пользователями, предоставляя информацию, отвечая на вопросы, выполняя задачи и предоставляя интерактивные сценарии.

### Анализ предметной области

При анализе предметной области веб-приложения "Конструктор чат-ботов" следует учитывать следующие аспекты:

**Функциональные требования:**

* **Создание ботов**: Пользователь должен иметь возможность создавать новые чат-боты для платформы Telegram.
* **Настройка ботов**: Пользователь должен иметь возможность настраивать параметры ботов, такие как имя, аватар, приветственное сообщение и команды.
* **Управление командами**: Пользователь должен иметь возможность настраивать команды, определять текстовые ответы или скрипты, которые будут выполняться при получении определенных команд от пользователя.
* **Создание сценариев**: Пользователь должен иметь возможность создавать интерактивные сценарии, определяющие последовательность вопросов и ответов для взаимодействия с пользователем.
* **Интеграция с внешними сервисами**: Пользователь должен иметь возможность интегрировать чат-ботов с различными внешними сервисами и API для получения и отправки данных.
* **Аналитика использования**: Пользователь должен иметь доступ к статистике использования чат-ботов, такой как количество подписчиков, количество запросов, популярные команды.

**Нефункциональные требования:**

* **Производительность**: Приложение должно обеспечивать высокую производительность для быстрого создания и настройки чат-ботов, а также эффективного обработки запросов от пользователей.
* **Масштабируемость**: Приложение должно быть масштабируемым, чтобы легко справляться с ростом числа пользователей и созданных чат-ботов.
* **Безопасность**: Приложение должно обеспечивать безопасность данных пользователей и настроек чат-ботов, а также предотвращать несанкционированный доступ к функциональности и административным возможностям.
* **Доступность**: Приложение должно быть доступным для пользователей на различных устройствах и с разными ограничениями, такими как доступность для людей с ограниченными возможностями.
* **Удобство использования интерфейса**: Приложение должно иметь интуитивно понятный и удобный пользовательский интерфейс, который позволяет легко создавать, настраивать и управлять чат-ботами.

**Интеграция с Telegram API:**

* **Интеграция с Telegram**: Приложение должно взаимодействовать с Telegram API для создания и настройки ботов, отправки и получения сообщений, управления подписчиками и других функций, предоставляемых Telegram API.

**Управление данными:**

* **Хранение данных**: Приложение должно предоставлять механизм для хранения данных о созданных ботах, настройках, командах, сценариях и подписчиках.
* **Безопасность данных**: Данные, хранящиеся в приложении, должны быть защищены с использованием соответствующих мер безопасности, таких как шифрование и аутентификация.

В процессе разработки веб-приложения необходимо уделить внимание требованиям пользователей, проанализировать возможности и ограничения Telegram API, выбрать подходящие технологии и инструменты разработки, а также рассмотреть вопросы безопасности и масштабируемости приложения.

### Выбор технологий для разработки

1. **React JS:**

React JS является популярной и мощной библиотекой для разработки пользовательского интерфейса. Она обеспечивает быструю и эффективную отрисовку компонентов, удобное управление состоянием приложения и простоту переиспользования кода. Благодаря виртуальному DOM, React JS обеспечивает быстрое обновление только измененных частей интерфейса.

1. **Axios:**

Axios является удобной библиотекой для выполнения HTTP-запросов в браузере или на сервере. В контексте "Конструктора чат-ботов", Axios позволяет установить связь между клиентской и серверной частями приложения. Он обеспечивает возможность отправки и получения данных.

1. **Django:**

Django является мощным фреймворком для разработки веб-приложений на языке Python. Он обеспечивает высокую производительность, безопасность и масштабируемость. В контексте "Конструктора чат-ботов", Django может быть использован для взаимодействия с базой данных и управления бизнес-логикой приложения.

1. **MSSQL:**

Выбор конкретной системы управления базами данных (СУБД) зависит от требований проекта, но MSSQL (Microsoft SQL Server) является одной из популярных и широко используемых СУБД. Она предоставляет надежное хранение данных, обеспечивает высокую производительность и масштабируемость. MSSQL может быть использована для хранения данных о пользователях, созданных чат-ботах, настройках, командах, сценариях и подписчиках

1. **C#:**

C# является мощным языком программирования, который может быть использован для разработки программной логики, поддерживающей работу чат-ботов. Он имеет богатую экосистему инструментов и библиотек для работы с API, обработки сообщений, выполнения бизнес-логики и интеграции с другими сервисами. Использование C# позволяет разрабатывать функциональность чат-ботов, такую как обработка команд, генерация ответов и взаимодействие с внешними сервисами.

Указанные технологии (React JS, Axios, Django, MSSQL и C#) были выбраны с учетом их преимуществ, популярности, поддержки сообществом разработчиков, соответствия требованиям проекта, знаний и опыта команды. Также они хорошо подходили под технологии, использующиеся в компании, для которой и реализовывался конструктор.

## Проектирование базы данных

Для регистрации пользователя необходимы его электронная почта, логин и пароль. Также к пользователю привязывается его идентификатор.

Для регистрации бота необходимы его идентификатор пользователя, имя, уникальное имя (идентификатор бота в Telegram) и токен подключения к боту. Также у бота есть поле отвечающие за статус бота (включён/выключен).

Должен вестись список подключённых пользователей к чат-ботам.

Для добавления команд боту необходима следующая информация: идентификатор бота, имя команды и тип команды. Тип команды берётся из справочника команд, в нём хранится идентификатор и название типа.

К каждой команде должна быть привязана запись из таблицы (сообщение/рассылка), соответствующей типу команды.

Для добавления сообщения необходимы идентификатор команды и сообщение.

Для добавления рассылки необходимы идентификатор команды, сообщение, дата и время рассылки.

Также к команде может быть привязана запись из таблицы «Медиа», в которой хранится идентификатор команды, имя файла, тип файла и сам файл в формате base64.

У команды может быть фразы вызова, которые заносятся в соответствующую таблицу.

Команды могут быть связаны с другими командами. Такие связи заносятся в таблицу связей следующим образом: идентификатор текущей команды и идентификатор следующей команды.

Структура базы данных изображена на рисунке 1.

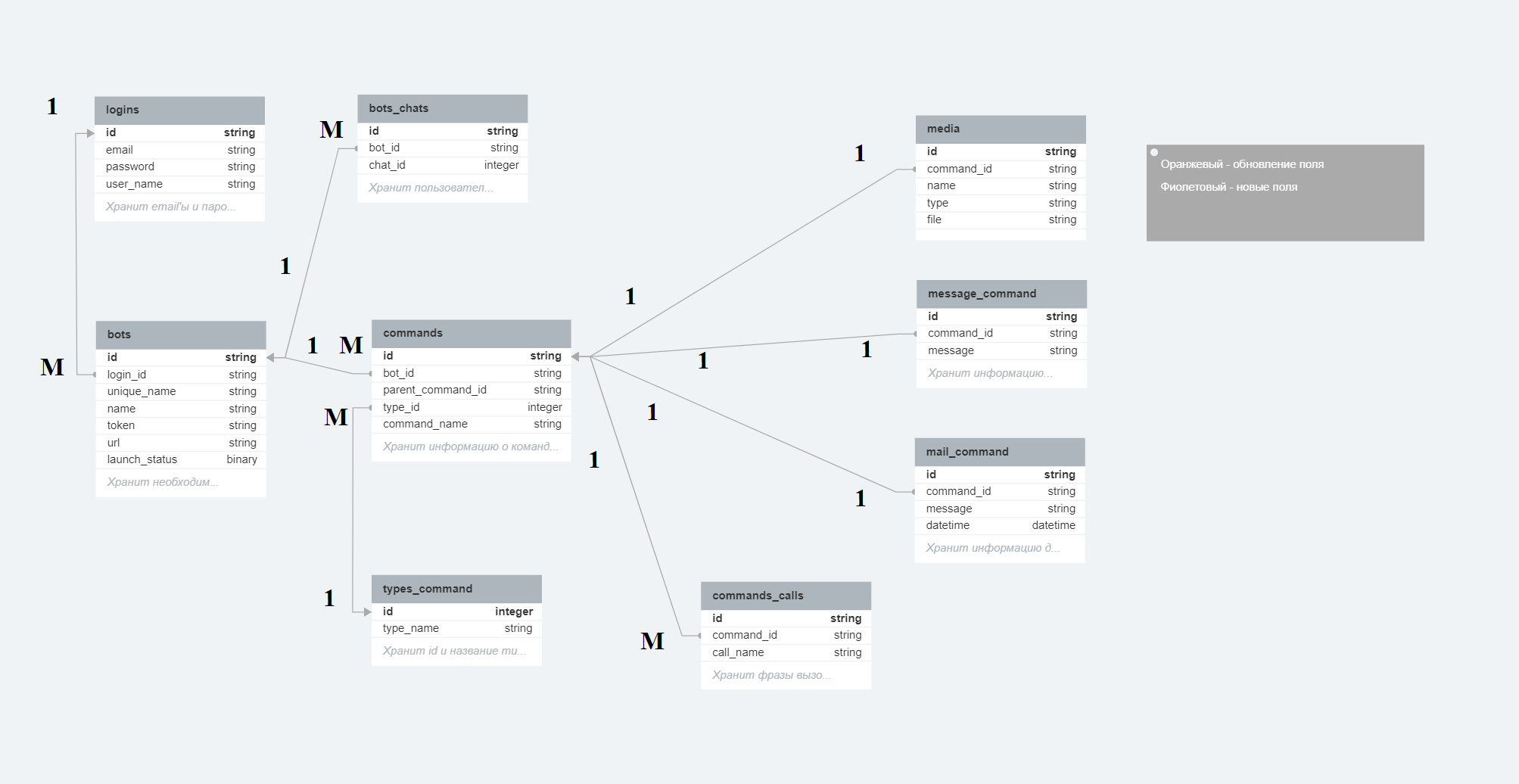


Рисунок 1 – Структура базы данных

Ниже приведен перевод полей таблиц:

Logins – регистрации

id – идентификатор;

email – электронная почта;

password – пароль;

user\_name – имя пользователя.

Bots – Боты

id – идентификатор;

login\_id – идентификатор пользователя;

unique\_name – имя-идентификатор бота;

name – имя бота;

token – ключ подключения бота;

url – адрес подключаемого бота;

launch\_status – состояние запуска.

Bots\_chats – чаты с ботами

id – идентификатор;

bot\_id – идентификатор бота;

chat\_id – идентификатор чата.

Commands – Боты

id – идентификатор;

bot\_id – идентификатор бота;

type\_id – идентификатор типа команды;

command\_name – название команды;

link\_status – статус связи команды.

Types\_command – типы команд

id – идентификатор;

type\_name – название типа.

Commands\_calls – вызовы команд

id – идентификатор;

command\_id – идентификатор команды;

call\_name – фраза вызова.

Link\_commands – связи команд

id – идентификатор;

current\_command – идентификатор текущей команды;

following\_command – идентификатор следующей команды.

Message\_command – команды сообщения

id – идентификатор;

command\_id – идентификатор команды;

message – сообщение.

Mail\_command – регистрации

id – идентификатор;

command\_id – идентификатор команды;

message – сообщение;

datetime – дата и время.

Commands – Боты

id – идентификатор;

command\_id – идентификатор команды;

name – название файла;

type – тип файла;

file – файл.

## Разработка дизайна приложения

Дизайн сайта является современным и минималистичным, с акцентом на простоту использования и интуитивную навигацию. Цветовая схема включает в себя цвета #393330, #FF941A, #EFEFEF, #FFFFFF.



Рисунок 1. Цветовая гамма дизайна сайта

В качестве шрифта использовался шрифт Formular.

Изображение выглядит как текст, снимок экрана, Шрифт, дизайн

Автоматически созданное описание

Рисунок 3.1 Шрифт рисунка сайта.

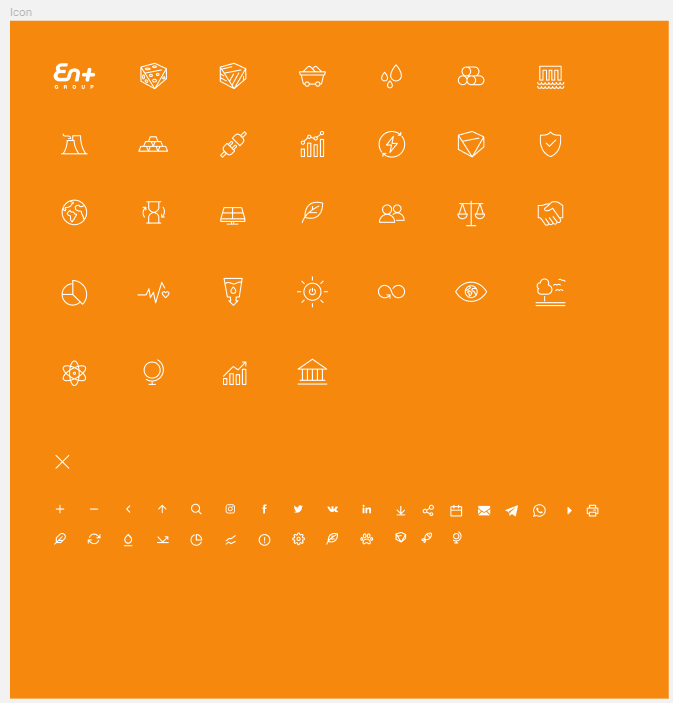


Рисунок 3.2 Иконки дизайна сайта.

Главная страница состоит из хедера с логотипом компании и основными навигационными элементами (кнопка входа и регистрации).

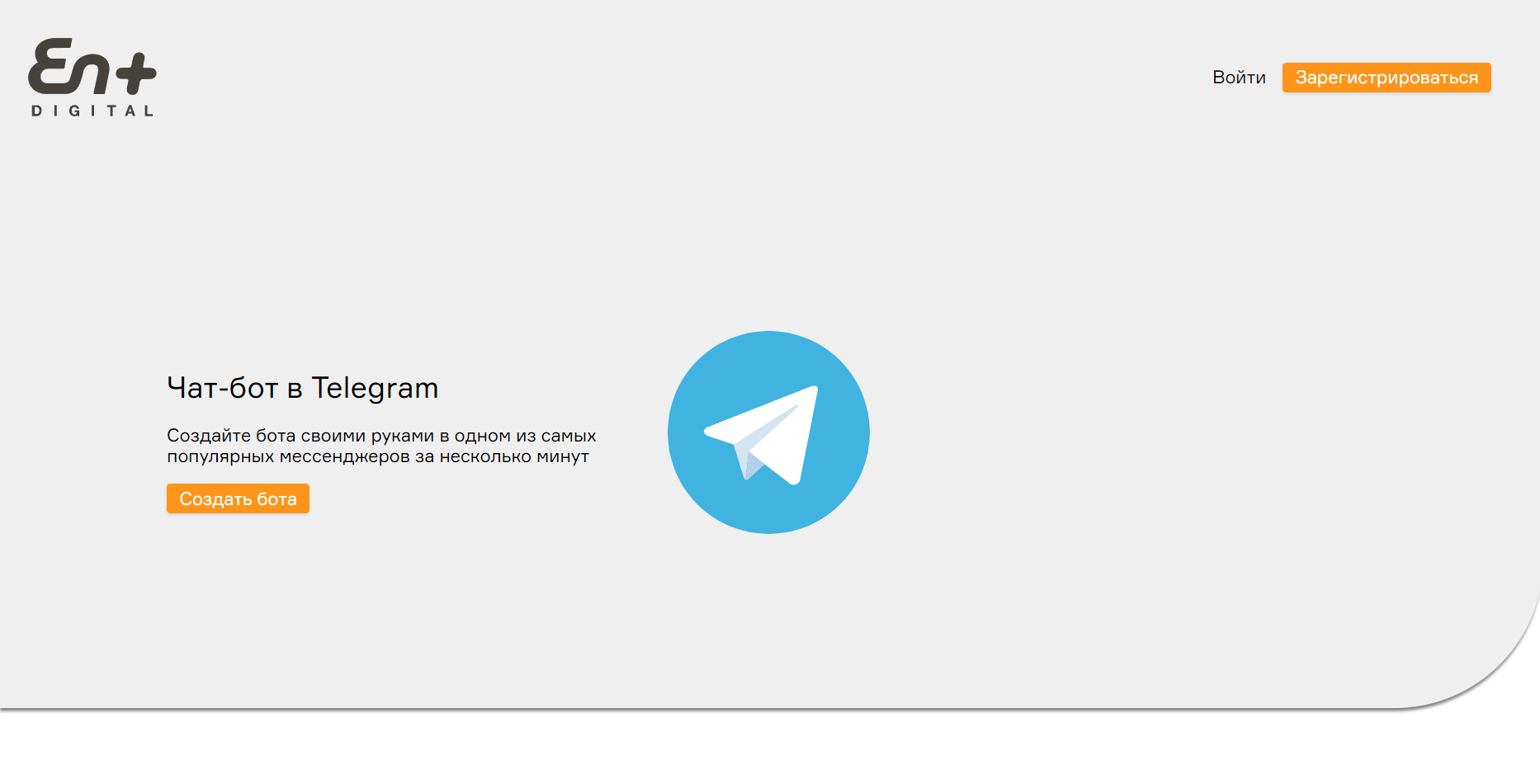


Рисунок 3.3 Стартовая страница сайта.

Изображение выглядит как текст, снимок экрана, Шрифт, дизайн

Автоматически созданное описание

Рисунок 3.4 Модальное окно авторизации.

Изображение выглядит как текст, снимок экрана, Шрифт, программное обеспечение

Автоматически созданное описание

Рисунок 3.5 Модальное окно регистрации

Далее после входа в систему появляется экран со списком ботов. Список ботов реализован в виде списка блоков, содержащего имя бота и кнопку, при нажатии на которую появляется окно с действиями. Также в верхней части экрана реализована шапка сайта (отдельный компонент на React), в правой части которой находится логотип копании, а слева email, иконка пользователя и выпадающее меню с действиями (выход из системы, открытие настроек).

Изображение выглядит как текст, снимок экрана, программное обеспечение

Автоматически созданное описание

Рисунок 3.6 Список ботов.

Изображение выглядит как текст, Шрифт, снимок экрана, белый

Автоматически созданное описание

Рисунок 3.7 Доступные действия для пользователя.

При нажатии на имя бота открывается конструктор, который реализован в виде холста, на котором расположены команды бота. В левой части экрана расположены блоки команд.

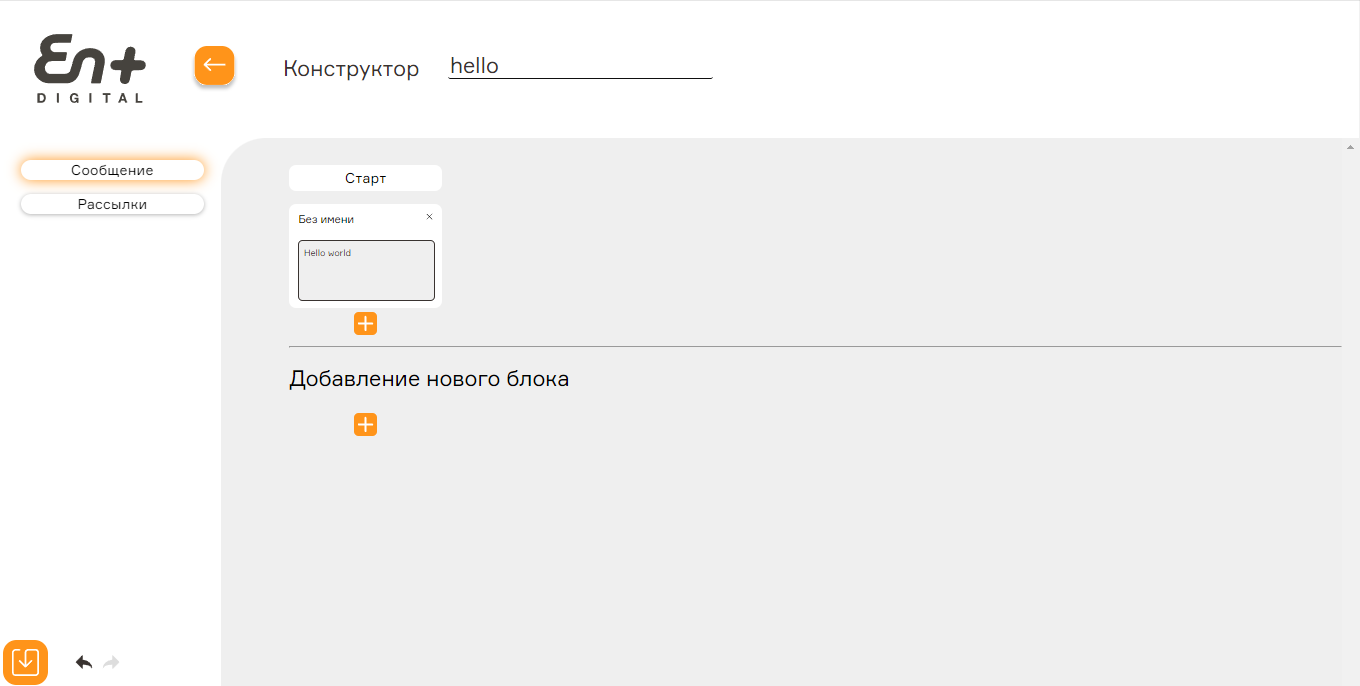


Рисунок 3.8 Конструктор чат-ботов

Изображение выглядит как текст, снимок экрана, программное обеспечение, Значок на компьютере

Автоматически созданное описание

Рисунок 3.9 Редактирование блока сообщения.

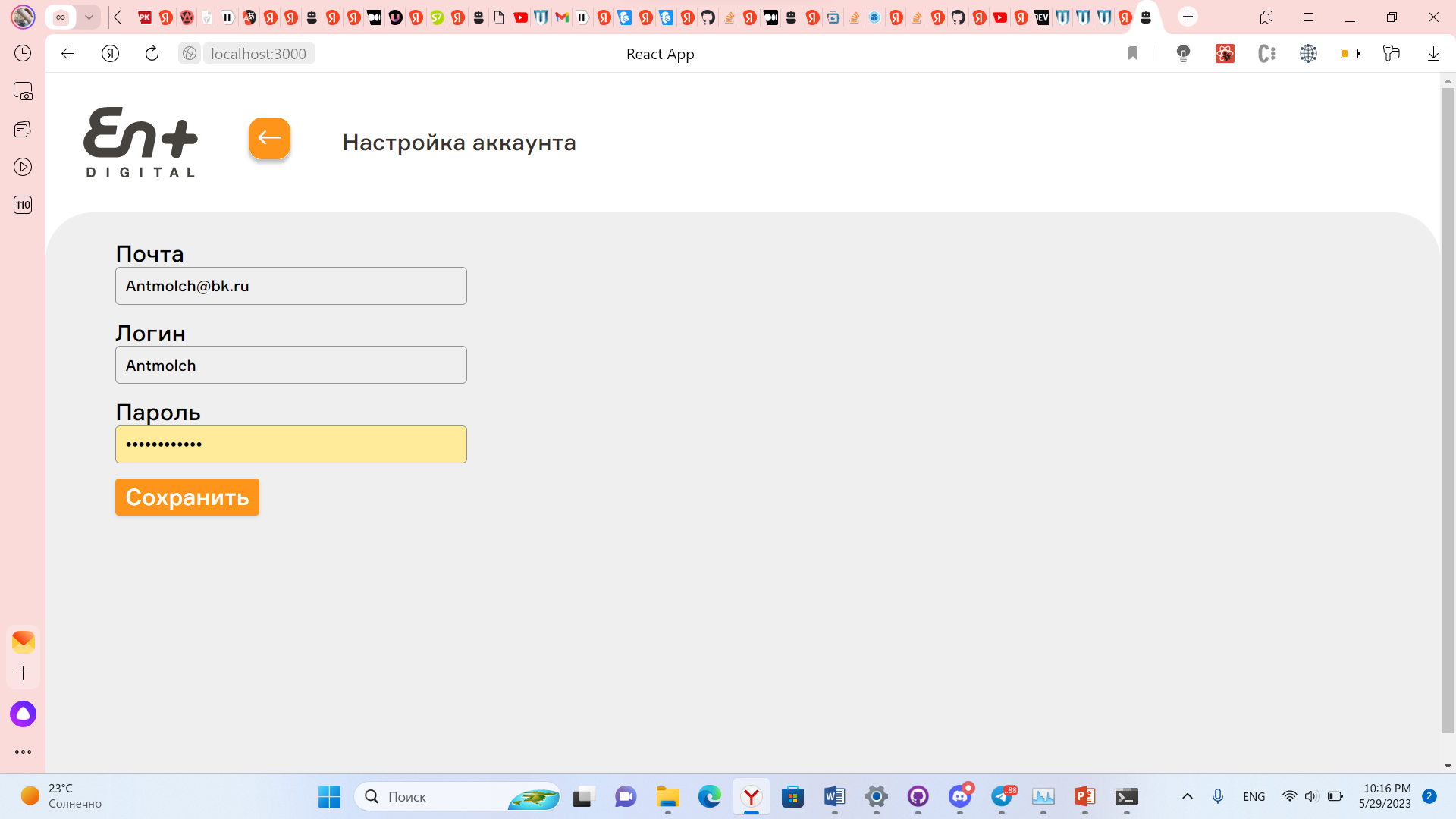


Рисунок 3.10 Настройки аккаунта.

При верстке использовался чистый CSS.

## Разработка серверной составляющей веб-приложения

Серверная часть веб-приложения, написанная на Django, является бэк-частью приложения, которая отвечает за обработку данных и взаимодействие с фронт-частью.

Django - это популярный фреймворк для разработки веб-приложений на языке программирования Python. Он предоставляет разработчикам мощные инструменты и готовые решения для создания полнофункциональных веб-приложений.

Для начала разработки серверной части приложения на Django необходимо настроить окружение, установить Django и другие необходимые зависимости. После этого можно приступить к созданию моделей, представлений и URL-маршрутов, которые составляют основу серверной логики приложения.

Модели в Django представляют сущности и их отношения в базе данных. Они определяются с использованием классов Python, которые наследуются от базового класса models.Model. В моделях определяются поля, связи с другими моделями и методы для работы с данными.

Представления (views) в Django определяют логику обработки запросов от клиента. Они могут быть функциональными или классовыми. В функциональных представлениях определяются функции, которые принимают запрос и возвращают ответ, основанный на данных из моделей. В классовых представлениях используется класс, который наследуется от базового класса views. View и имеет методы для обработки различных типов запросов.

URL-маршруты определяют, какие представления должны быть вызваны для обработки определенных URL-адресов. Они могут быть определены в файле urls.py, где указывается соответствие между URL-шаблонами и представлениями.

В Django также доступны множество дополнительных возможностей и инструментов, таких как аутентификация, авторизация, работа с формами, обработка файлов и многое другое. Django также обладает мощной административной панелью, которая позволяет управлять данными и настройками приложения.

Разработка серверной части на Django позволяет создавать мощные и надежные веб-приложения, обеспечивая удобные инструменты и готовые решения для различных аспектов разработки.

Бэкэнд часть приложения включает в себя следующие основные элементы:

1. Маршрутизация: Для обработки запросов и определения маршрутов можно использовать фреймворк Django, который предоставляет мощный механизм маршрутизации. Маршруты определяются в файле urls.py и связываются с соответствующими представлениями.
2. Представления (views): Представления в Django определяют логику обработки запросов от клиента. Они могут включать в себя операции чтения, создания, обновления и удаления данных. Представления могут работать с моделями, выполнять бизнес-логику, обращаться к базе данных и возвращать данные в формате JSON или HTML.
3. Модели и базы данных: В Django можно определять модели, которые представляют сущности и их отношения в базе данных. Django обеспечивает интеграцию с различными СУБД, такими как PostgreSQL, MySQL, SQLite и другими. Модели позволяют выполнять операции CRUD (создание, чтение, обновление, удаление) с данными.
4. Сериализаторы: В Django используются для преобразования сложных типов данных, таких как модели базы данных, в форматы, которые можно легко передавать по сети или использовать в различных контекстах.
5. Аутентификация и авторизация: Django предоставляет инструменты для аутентификации пользователей и управления доступом к ресурсам. Встроенная аутентификация позволяет регистрировать пользователей, выполнять вход в систему, управлять сессиями и проверять права доступа.
6. Взаимодействие с другими сервисами и API: Django может взаимодействовать с внешними сервисами и API. Это позволяет интегрировать приложение с различными сервисами, такими как платежные системы, социальные сети, внешние базы данных и другие.
7. Тестирование: Django предоставляет механизмы для написания и выполнения тестов. Тестирование позволяет убедиться в корректности работы приложения, обнаружить ошибки и поддерживать стабильность функциональности.
8. Логирование: Django обеспечивает механизмы для логирования событий и ошибок, что помогает в отслеживании и исправлении проблем в приложении.
9. Обработка файлов: Django предоставляет средства для загрузки, хранения и обработки файлов. Это позволяет работать с изображениями, документами и другими типами файлов.
10. Работа с формами: Django предоставляет инструменты для создания и обработки форм. Это включает валидацию данных, отображение форм на страницах, обработку отправленных данных и сохранение в базе данных.

### 4.1 Описание маршрутов

Бэк часть представляет отдельные приложения:

1. index(первая версия api)
2. enDjangoProject(основная часть сайта содержащая его настройки)
3. api(рабочая версия api)

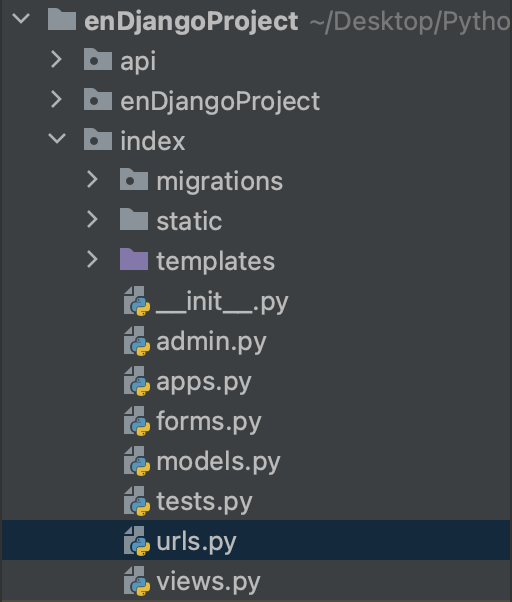


Рисунок 4.4.1 Топология файлов бэк части

Точка входа - enDjangoProject/urls.py, которая имеет следующие ссылки

urlpatterns = [  
 path('admin/', admin.site.urls),  
 path('',include('index.urls')),  
 path('api/',include("api.urls")),  
  
  
 path('auth/', include('djoser.urls')),  
 path('auth/', include('djoser.urls.authtoken')),  
 path('auth/', include('djoser.urls.jwt')),  
]

В данном коде определяются маршруты (URL-пути) приложения Django.

1. path('admin/', admin.site.urls): Этот путь отвечает за административный интерфейс Django. По этому URL пользователь будет иметь доступ к админ-панели приложения.
2. path('',include('index.urls')): Этот путь относится к приложению "index". Он определяет корневой URL-путь, который обрабатывается при помощи маршрутов, определенных в файле index.urls.
3. path('api/',include("api.urls")): Этот путь относится к приложению "api". Он определяет URL-префикс /api/, который будет использоваться для всех API-маршрутов, определенных в файле api.urls.
4. path('auth/', include('djoser.urls')): Этот путь относится к аутентификации и регистрации пользователей. Он использует URL-префикс /auth/ и включает маршруты, предоставляемые библиотекой Djoser для обработки операций аутентификации и регистрации.
5. path('auth/', include('djoser.urls.authtoken')): Этот путь относится к генерации и управлению токенами аутентификации. Он использует URL-префикс /auth/ и включает маршруты, предоставляемые Djoser для работы с токенами аутентификации.
6. path('auth/', include('djoser.urls.jwt')): Этот путь относится к JSON Web Token (JWT) аутентификации. Он использует URL-префикс /auth/ и включает маршруты, предоставляемые Djoser для работы с JWT.

Описание api/urls.py

urlpatterns = [  
 path('getuser', UserInfoAPIView.as\_view()),  
  
 path('bot/create', BotCreateView.as\_view()),  
 path('bot/all', BotsListView.as\_view()),  
 path('bot/<str:pk>', BotDetailView.as\_view(), name='bot-detail'),  
 # path('bot/', BotAPIView.as\_view(), name='bot'),  
  
 path('commandType/create', TypeCommandCreateView.as\_view()),  
 path('commandType/all', TypeCommandListView.as\_view()),  
 path('commandType/<str:pk>', CommandTypeDetailView.as\_view()),  
  
 path('command/create', CommandCreateView.as\_view()),  
 path('command/all', CommandsListView.as\_view()),  
 path('command/<str:pk>', CommandDetailView.as\_view()),  
  
 path('command/call/all', CallsListView.as\_view()),  
 path('command/call/create', CallsCreateView.as\_view()),  
 path('command/call/<str:pk>', CallsDetailView.as\_view()),  
  
 path('media/create',MediaCreateView.as\_view()),  
 path('media/all',MediaListView.as\_view()),  
 path('media/<str:pk>',MediaDetailView.as\_view()),  
  
 path('messageCommand/create', MessageCommandCreateView.as\_view()),  
 path('messageCommand/all', MessageCommandListView.as\_view()),  
 path('messageCommand/<str:pk>', MessageCommandDetailView.as\_view()),  
  
 path('mailCommand/create', MailCommandCreateView.as\_view()),  
 path('mailCommand/all', MailCommandListView.as\_view()),  
 path('mailCommand/<str:pk>', MailCommandDetailView.as\_view()),  
  
 path('botChat/create', BotChatCreateView.as\_view()),  
 path('botChat/all', BotChatListView.as\_view()),  
 path('botChat/<str:pk>', BotChatDetailView.as\_view()),  
  
 path('link/create', LinkCreateView.as\_view()),  
 path('link/all', LinkListView.as\_view()),  
 path('link/<str:pk>', LinkDetailView.as\_view()),

]

Ниже представлен краткий обзор каждого пути:

1. path('getuser', UserInfoAPIView.as\_view()): Обработчик этого пути представляет представление UserInfoAPIView, которое отвечает за получение информации о пользователе.
2. path('bot/create', BotCreateView.as\_view()): Путь для создания нового бота с использованием представления BotCreateView.
3. path('bot/all', BotsListView.as\_view()): Путь для получения списка всех ботов с использованием представления BotsListView.
4. path('bot/<str:pk>', BotDetailView.as\_view(), name='bot-detail'): Путь для получения деталей определенного бота с использованием представления BotDetailView. Параметр <str:pk> указывает на идентификатор бота.
5. path('commandType/create', TypeCommandCreateView.as\_view()): Путь для создания нового типа команды с использованием представления TypeCommandCreateView.
6. path('commandType/all', TypeCommandListView.as\_view()): Путь для получения списка всех типов команд с использованием представления TypeCommandListView.
7. path('commandType/<str:pk>', CommandTypeDetailView.as\_view()): Путь для получения деталей определенного типа команды с использованием представления CommandTypeDetailView. Параметр <str:pk> указывает на идентификатор типа команды.
8. path('command/create', CommandCreateView.as\_view()): Путь для создания новой команды с использованием представления CommandCreateView.
9. path('command/all', CommandsListView.as\_view()): Путь для получения списка всех команд с использованием представления CommandsListView.
10. path('command/<str:pk>', CommandDetailView.as\_view()): Путь для получения деталей определенной команды с использованием представления CommandDetailView. Параметр <str:pk> указывает на идентификатор команды.

И так далее. Каждый путь определяет маршрут и соответствующее представление для обработки запросов на определенный URL-шаблон. Это позволяет организовать обработку запросов на различные операции с данными в приложении, такие как создание, чтение, обновление и удаление.

### 4.2 Описание моделей

class Bot(models.Model):  
 id = models.UUIDField(primary\_key=True, default=uuid.uuid4, editable=False)  
 login\_id = models.ForeignKey(User, max\_length=255, default=None, on\_delete=models.CASCADE)  
 unique\_name = models.CharField(max\_length=255, default=None)  
 name = models.CharField(max\_length=255, default=None)  
 token = models.CharField(max\_length=255, default=None)  
 url = models.CharField(max\_length=255, default="")  
 launch\_status = models.BooleanField(max\_length=255,default=None)  
  
 def \_\_str\_\_(self):  
 return self.name  
  
class TypeCommand(models.Model):  
 id = models.UUIDField(primary\_key=True, default=uuid.uuid4, editable=False)  
 name = models.CharField(max\_length=255, default=None)  
  
 def \_\_str\_\_(self):  
 return self.name  
  
  
 def save(self, \*args, \*\*kwargs):  
 if self.pk is None: # Проверяем, что объект не имеет первичного ключа (т.е. новый объект)  
 raise ValidationError("Нельзя добавлять новые объекты TypeCommand.")  
 super().save(\*args, \*\*kwargs)  
  
@receiver(post\_migrate)  
def create\_default\_type\_commands(sender, \*\*kwargs):  
 if sender.name == 'api':  
 TypeCommand.objects.get\_or\_create(name='mail')  
 TypeCommand.objects.get\_or\_create(name='message')  
  
class Command(models.Model):  
 id = models.UUIDField(primary\_key=True, default=uuid.uuid4, editable=False)  
 bot\_id = models.ForeignKey(Bot, on\_delete=models.CASCADE)  
 type\_id = models.ForeignKey(TypeCommand, on\_delete=models.CASCADE, null=False, default=None)  
 name = models.CharField(max\_length=255,default=None)  
 link\_status = models.BooleanField(default=False)  
  
 def \_\_str\_\_(self):  
 return self.name  
  
class CommandCall(models.Model):  
 id = models.UUIDField(primary\_key=True, default=uuid.uuid4, editable=False)  
 command\_id = models.ForeignKey(Command, on\_delete=models.CASCADE,related\_name='calls')  
 name = models.CharField(max\_length=255,default=None)  
  
class Media(models.Model):  
 id = models.UUIDField(primary\_key=True, default=uuid.uuid4, editable=False)  
 command\_id = models.OneToOneField(Command, on\_delete=models.CASCADE,related\_name='media')  
 name = models.CharField(max\_length=255,default=None)  
 type = models.CharField(max\_length=255,default=None)  
 file = models.TextField()  
  
class MessageCommand(models.Model):  
 id = models.UUIDField(primary\_key=True, default=uuid.uuid4, editable=False)  
 command\_id = models.OneToOneField(Command, on\_delete=models.CASCADE,related\_name='messageCommand')  
 message = models.TextField()  
  
class MailCommand(models.Model):  
 id = models.UUIDField(primary\_key=True, default=uuid.uuid4, editable=False)  
 command\_id = models.OneToOneField(Command, on\_delete=models.CASCADE,related\_name='mailCommand')  
 message = models.TextField()  
 datetime = models.DateTimeField()  
  
class BotChat(models.Model):  
 id = models.UUIDField(primary\_key=True, default=uuid.uuid4, editable=False)  
 bot\_id = models.ForeignKey(Bot, on\_delete=models.CASCADE)  
 chat\_id = models.CharField(max\_length=32)  
  
class LinkCommand(models.Model):  
 id = models.UUIDField(primary\_key=True, default=uuid.uuid4, editable=False)  
 current = models.ForeignKey(Command, on\_delete=models.CASCADE, related\_name='current')  
 follow = models.ManyToManyField(Command, related\_name='follow')

Приведенный код описывает модели для курсовой работы. Вот краткое описание каждой модели:

1. Bot - модель для представления ботов. Она содержит поля, такие как id, login\_id, unique\_name, name, token, url и launch\_status.
2. TypeCommand - модель для представления типов команд. Она содержит поле id и name.
3. Command - модель для представления команд. Она содержит поля, такие как id, bot\_id, type\_id, name и link\_status.
4. CommandCall - модель для представления вызовов команды. Она содержит поля id, command\_id и name.
5. Media - модель для представления медиафайлов, связанных с командами. Она содержит поля id, command\_id, name, type и file.
6. MessageCommand - модель для представления сообщений, связанных с командами. Она содержит поля id, command\_id и message.
7. MailCommand - модель для представления почтовых команд. Она содержит поля id, command\_id, message и datetime.
8. BotChat - модель для представления чатов ботов. Она содержит поля id, bot\_id и chat\_id.
9. LinkCommand - модель для представления связи между командами. Она содержит поля id, current и follow, где current - текущая команда, а follow - связанные команды.

Каждая модель имеет свой первичный ключ id типа UUIDField. Некоторые модели также содержат внешние ключи, которые связывают модели между собой.

### 4.3 Описание сериализаторов

class CommandCallListSerializer(serializers.ModelSerializer):  
 class Meta:  
 model = CommandCall  
 fields = "\_\_all\_\_"  
  
class CommandCallDetailSerializer(serializers.ModelSerializer):  
 class Meta:  
 model = CommandCall  
 fields = ('id', 'name','command\_id')  
  
 def validate(self, data):if not data.get('command\_id'):  
 raise serializers.ValidationError("Поле commmand\_id обязательно для заполнения")  
  
 if not data.get('name'):  
 raise serializers.ValidationError("Поле name обязательно для заполнения")  
 return data  
  
  
class BotDetailSerializer(serializers.ModelSerializer):  
  
 # commands = CommandListSerializer(many=True, read\_only=True)  
  
 class Meta:  
 model = Bot  
 fields = ('id','unique\_name','name','token','url','launch\_status')  
  
 def validate(self, data):  
 *"""  
 Проверяем, что все обязательные поля заполнены  
 """* if not data.get('unique\_name'):  
 raise serializers.ValidationError("Поле unique\_name обязательно для заполнения")  
  
 if not data.get('name'):  
 raise serializers.ValidationError("Поле name обязательно для заполнения")  
  
 if not data.get('token'):  
 raise serializers.ValidationError("Поле token обязательно для заполнения")  
  
 # if not data.get('url'):  
 # raise serializers.ValidationError("Поле url обязательно для заполнения")  
  
 return data  
  
class TypeCommandDetailSerializer(serializers.ModelSerializer):  
 class Meta:  
 model = TypeCommand  
 fields = ('id', 'name')  
  
 def validate(self, data):  
 *"""  
 Проверяем, что все обязательные поля заполнены  
 """* if not data.get('name'):  
 raise serializers.ValidationError("Поле name обязательно для заполнения")  
 return data  
  
  
class MediaDetailSerializer(serializers.ModelSerializer):  
 class Meta:  
 model = Media  
 fields = ('id','command\_id','name','type','file')  
  
  
class CommandsListSerializer(serializers.ModelSerializer):  
 class Meta:  
 model = Command  
 fields = "\_\_all\_\_"  
  
 def to\_representation(self, instance):  
 representation = super().to\_representation(instance)  
 type\_ids = representation.pop('type\_id', [])  
 types = TypeCommand.objects.filter(id\_\_in=type\_ids)  
 type\_names = TypeCommandDetailSerializer(types, many=True).data  
 representation['type'] = type\_names  
  
 command\_calls = instance.calls.all()  
 call\_data = CommandCallDetailSerializer(command\_calls, many=True).data  
 representation['calls'] = call\_data  
  
 media = Media.objects.filter(command\_id=instance)  
 media\_data = MediaDetailSerializer(media, many=True).data  
 representation['media'] = media\_data  
  
 message\_command = MessageCommand.objects.filter(command\_id=instance)  
 message\_command\_data = MessageCommandDetailSerializer(message\_command, many=True).data  
 representation['message\_command'] = message\_command\_data  
  
 mail\_command = MailCommand.objects.filter(command\_id=instance)  
 mail\_command\_data = MailCommandDetailSerializer(mail\_command, many=True).data  
 representation['mail\_command'] = mail\_command\_data  
  
 link = LinkCommand.objects.filter(current=instance)  
 link\_data = LinkDetailSerializer(link, many=True).data  
 representation['link'] = link\_data  
  
 return representation  
  
  
class BotsListSerializer(serializers.ModelSerializer):  
 commands = serializers.SerializerMethodField()  
 chat = serializers.SerializerMethodField()  
 class Meta:  
 model = Bot  
 fields = "\_\_all\_\_"  
  
  
 def get\_chat(self, bot):  
 chats = BotChat.objects.filter(bot\_id=bot.id)  
 return BotChatDetailSerializer(chats, many=True).data  
  
 def get\_commands(self, bot):  
 commands = Command.objects.filter(bot\_id=bot.id)  
  
 return CommandDetailSerializer(commands, many=True).data  
  
  
 def \_\_init\_\_(self, \*args, \*\*kwargs):  
 request = kwargs['context']['request']  
 userid = get\_user\_id\_from\_request(request)  
 user = User.objects.get(id=userid)  
 super().\_\_init\_\_(\*args, \*\*kwargs)  
  
 self.Meta.queryset = Bot.objects.filter(login\_id=user.id)  
  
  
class TypeCommandListSerializer(serializers.ModelSerializer):  
 class Meta:  
 model = TypeCommand  
 fields = ('id', 'name')  
  
  
class CommandDetailSerializer(serializers.ModelSerializer):  
 message\_commands = serializers.SerializerMethodField()  
 mail\_commands = serializers.SerializerMethodField()  
 calls = serializers.SerializerMethodField()  
 media = serializers.SerializerMethodField()  
 links = serializers.SerializerMethodField()  
 class Meta:  
 model = Command  
 fields = ('id', 'name', 'link\_status', 'bot\_id', 'type\_id','message\_commands','mail\_commands','calls', 'media','links')  
  
 def get\_message\_commands(self, obj):  
 message\_commands = MessageCommand.objects.filter(command\_id=obj.id)  
 return MessageCommandDetailSerializer(message\_commands, many=True).data  
  
 def get\_mail\_commands(self,obj):  
 mail\_commands = MailCommand.objects.filter(command\_id=obj.id)  
 return MailCommandDetailSerializer(mail\_commands, many=True).data  
  
 def get\_calls(self,obj):  
 calls = CommandCall.objects.filter(command\_id=obj.id)  
 return CommandCallDetailSerializer(calls, many=True).data  
  
 def get\_media(self, obj):  
 calls = Media.objects.filter(command\_id=obj.id)  
 return MediaDetailSerializer(calls, many=True).data  
  
 def get\_links(self,obj):  
 links = LinkCommand.objects.filter(current\_id=obj.id)  
 return LinkDetailSerializer(links, many=True).data  
  
class MessageCommandDetailSerializer(serializers.ModelSerializer):  
 class Meta:  
 model = MessageCommand  
 fields = ('id','command\_id', 'message')  
  
  
class MailCommandDetailSerializer(serializers.ModelSerializer):  
 class Meta:  
 model = MailCommand  
 fields = ('id','command\_id','message','datetime')  
  
class BotChatDetailSerializer(serializers.ModelSerializer):  
 class Meta:  
 model = BotChat  
 fields = ("id","bot\_id","chat\_id")  
  
class LinkDetailSerializer(serializers.ModelSerializer):  
 class Meta:  
 model = LinkCommand  
 fields = ("id","current","follow")

class CustomUserCreateSerializer(UserCreateSerializer):  
 def validate\_email(self, value):  
 if User.objects.filter(email=value).exists():  
 raise serializers.ValidationError('Email already exists')  
 return value  
  
 class Meta:  
 model = User  
 fields = ('id', 'email', 'password','username')  
  
class CustomTokenSerializer(TokenSerializer):  
 user\_id = serializers.IntegerField(source='user.id')  
  
 class Meta(TokenSerializer.Meta):  
 fields = TokenSerializer.Meta.fields + ('user\_id',)  
  
  
class BotUUIDSerializer(serializers.Serializer):  
 uuid = serializers.UUIDField()  
  
 def validate\_uuid(self, value):  
 try:  
 uuid\_obj = UUID(value)  
 except (TypeError, ValueError):  
 raise serializers.ValidationError("Invalid UUID format")  
  
 return value

Краткое описание сериализаторов из предоставленного кода:

1. CommandCallListSerializer: Сериализатор для модели CommandCall, используется для представления списка объектов CommandCall.
2. CommandCallDetailSerializer: Сериализатор для модели CommandCall, используется для детального представления объекта CommandCall с определенными полями.
3. BotDetailSerializer: Сериализатор для модели Bot, используется для детального представления объекта Bot с определенными полями.
4. TypeCommandDetailSerializer: Сериализатор для модели TypeCommand, используется для детального представления объекта TypeCommand с определенными полями.
5. MediaDetailSerializer: Сериализатор для модели Media, используется для детального представления объекта Media с определенными полями.
6. CommandsListSerializer: Сериализатор для модели Command, используется для представления списка объектов Command с определенными полями.
7. BotsListSerializer: Сериализатор для модели Bot, используется для представления списка объектов Bot с определенными полями.
8. TypeCommandListSerializer: Сериализатор для модели TypeCommand, используется для представления списка объектов TypeCommand.
9. CommandDetailSerializer: Сериализатор для модели Command, используется для детального представления объекта Command с определенными полями.
10. MessageCommandDetailSerializer: Сериализатор для модели MessageCommand, используется для детального представления объекта MessageCommand с определенными полями.
11. MailCommandDetailSerializer: Сериализатор для модели MailCommand, используется для детального представления объекта MailCommand с определенными полями.
12. BotChatDetailSerializer: Сериализатор для модели BotChat, используется для детального представления объекта BotChat с определенными полями.
13. LinkDetailSerializer: Сериализатор для модели LinkCommand, используется для детального представления объекта LinkCommand с определенными полями.
14. CustomUserCreateSerializer: Сериализатор для модели пользователя (User), используется для создания нового пользователя с определенными полями.
15. CustomTokenSerializer: Сериализатор для токена аутентификации, используется для представления токена с дополнительным полем user\_id.
16. BotUUIDSerializer: Сериализатор для UUID, используется для валидации и представления UUID.

### 4.4 Функция обработчик

@receiver(post\_migrate)  
def create\_superuser(sender, \*\*kwargs):  
 if User.objects.filter(username='admin').exists():  
 return  
 User.objects.create\_superuser('admin', 'admin@example.com', 'password')

Функция create\_superuser в коде, привязанном к сигналу post\_migrate, служит для автоматического создания суперпользователя при выполнении миграций в Django. Зачастую в процессе разработки и развертывания веб-приложения требуется наличие суперпользователя для административного доступа к системе. Создание суперпользователя вручную может быть неудобным и требовать дополнительных шагов.

При использовании данной функции и сигнала post\_migrate суперпользователь будет автоматически создан, если пользователь с именем "admin" отсутствует в базе данных. Это обеспечивает наличие суперпользователя с заранее заданными учетными данными (в данном случае "admin", "admin@example.com" и "password") для доступа к административной панели или другим привилегированным функциям приложения.

Такой подход упрощает развертывание и инициализацию приложения, поскольку необходимость вручную создавать суперпользователя отпадает, и суперпользователь будет автоматически создан при каждом выполнении миграций, если его еще нет.

### 4.5 Описание представлений

class UserInfoAPIView(APIView):  
 permission\_classes = [IsAuthenticated]  
  
 def get(self, request):  
 user = request.user  
 data = {  
 'username': user.username,  
 'email': user.email  
 }  
 return Response(data)  
  
class BotCreateView(generics.CreateAPIView):  
 serializer\_class = BotDetailSerializer  
  
 def perform\_create(self, serializer):  
 # получаем пользователя из запроса  
 request = self.request  
 userid = get\_user\_id\_from\_request(request)  
  
 user = self.request.user  
 print(userid)  
 # добавляем пользователя в словарь данных для сохранения объекта  
 serializer.validated\_data['login\_id'] = User.objects.get(id=userid)  
 # сохраняем объект  
 serializer.save()

class BotsListView(generics.ListAPIView):  
 serializer\_class = BotsListSerializer  
  
 def get\_queryset(self):  
 user\_id = get\_user\_id\_from\_request(self.request)  
 queryset = Bot.objects.filter(login\_id=user\_id)  
 return queryset  
  
  
class TypeCommandListView(generics.ListAPIView):  
 permission\_classes = [permissions.IsAuthenticated]  
 serializer\_class = TypeCommandListSerializer  
 queryset = TypeCommand.objects.all()  
  
class CommandsListView(generics.ListAPIView):  
 permission\_classes = [permissions.IsAuthenticated]  
 serializer\_class = CommandsListSerializer  
 queryset = Command.objects.all()  
  
class CommandCreateView(generics.CreateAPIView):  
 permission\_classes = [permissions.IsAuthenticated]  
 serializer\_class = CommandDetailSerializer

class CallsListView(generics.ListAPIView):  
 queryset = CommandCall.objects.all()  
 serializer\_class = CommandCallListSerializer  
  
class CallsCreateView(generics.CreateAPIView):  
 permission\_classes = [permissions.IsAuthenticated]  
 serializer\_class = CommandCallDetailSerializer  
  
class MediaCreateView(generics.CreateAPIView):  
 permission\_classes = [permissions.IsAuthenticated]  
 serializer\_class = MediaDetailSerializer  
  
  
class MediaListView(generics.ListAPIView):  
  
 permission\_classes = [permissions.IsAuthenticated]  
 queryset = Media.objects.all()  
 serializer\_class = MediaDetailSerializer  
  
  
class MessageCommandCreateView(generics.CreateAPIView):  
 permission\_classes = [permissions.IsAuthenticated]  
 serializer\_class = MessageCommandDetailSerializer  
  
class MessageCommandListView(generics.ListAPIView):  
 permission\_classes = [permissions.IsAuthenticated]  
 queryset = MessageCommand.objects.all()  
 serializer\_class = MessageCommandDetailSerializer  
  
class MailCommandCreateView(generics.CreateAPIView):  
 permission\_classes = [permissions.IsAuthenticated]  
 serializer\_class = MailCommandDetailSerializer  
  
  
class MailCommandListView(generics.ListAPIView):  
 permission\_classes = [permissions.IsAuthenticated]  
 queryset = MailCommand.objects.all()  
 serializer\_class = MailCommandDetailSerializer  
  
  
class BotChatCreateView(generics.CreateAPIView):  
 permission\_classes = [permissions.IsAuthenticated]  
 serializer\_class = BotChatDetailSerializer  
  
  
class BotChatListView(generics.ListAPIView):  
 permission\_classes = [permissions.IsAuthenticated]  
 queryset = BotChat.objects.all()  
 serializer\_class = BotChatDetailSerializer  
  
  
class LinkCreateView(generics.CreateAPIView):  
 permission\_classes = [permissions.IsAuthenticated]  
 serializer\_class = LinkDetailSerializer  
  
class LinkListView(generics.ListAPIView):  
 permission\_classes = [permissions.IsAuthenticated]  
 queryset = LinkCommand.objects.all()  
 serializer\_class = LinkDetailSerializer  
  
class CommandDetailView(generics.RetrieveUpdateDestroyAPIView):  
 #permission\_classes = [permissions.IsAuthenticated]  
 serializer\_class = CommandDetailSerializer  
 queryset = Command.objects.all()  
  
  
class CommandTypeDetailView(generics.RetrieveUpdateDestroyAPIView):  
 serializer\_class = TypeCommandDetailSerializer  
 queryset = TypeCommand.objects.all()  
  
class CallsDetailView(generics.RetrieveUpdateDestroyAPIView):  
 serializer\_class = CommandCallDetailSerializer  
 queryset = CommandCall.objects.all()  
  
class MediaDetailView(generics.RetrieveUpdateDestroyAPIView):  
 serializer\_class = MediaDetailSerializer  
 queryset = Media.objects.all()  
  
class MessageCommandDetailView(generics.RetrieveUpdateDestroyAPIView):  
 serializer\_class = MessageCommandDetailSerializer  
 queryset = MessageCommand.objects.all()  
  
class MailCommandDetailView(generics.RetrieveUpdateDestroyAPIView):  
 serializer\_class = MailCommandDetailSerializer  
 queryset = MailCommand.objects.all()  
  
class BotChatDetailView(generics.RetrieveUpdateDestroyAPIView):  
 serializer\_class = BotChatDetailSerializer  
 queryset = BotChat.objects.all()class BotDetailView(generics.RetrieveUpdateDestroyAPIView):  
 serializer\_class = BotUUIDSerializer  
 permission\_classes = [permissions.IsAuthenticated]  
 serializer\_class = BotDetailSerializer  
 queryset = Bot.objects.all()  
  
 def delete(self, request, \*args, \*\*kwargs):  
 instance = self.get\_object()  
 self.perform\_destroy(instance)  
 return Response({'message': 'Bot deleted successfully'})  
  
 def put(self, request, \*args, \*\*kwargs):  
 partial = kwargs.pop('partial', False)  
 instance = self.get\_object()  
 serializer = self.get\_serializer(instance, data=request.data, partial=partial)  
 serializer.is\_valid(raise\_exception=True)  
 self.perform\_update(serializer)  
 return Response(serializer.data)  
  
 def patch(self, request, \*args, \*\*kwargs):  
 return self.put(request, \*args, \*\*kwargs)

class LinkDetailView(generics.RetrieveUpdateDestroyAPIView):  
 serializer\_class = LinkDetailSerializer  
 queryset = LinkCommand.objects.all()

class TypeCommandCreateView(generics.CreateAPIView):  
 permission\_classes = [permissions.IsAuthenticated]  
 serializer\_class = TypeCommandDetailSerializer

Данный код представляет набор классов и представлений (views) для работы с API в Django, используя библиотеку Django REST Framework.

1. `UserInfoAPIView` - класс представления, который требует аутентификации пользователя (используется класс `IsAuthenticated`). При отправке GET-запроса на данный эндпоинт, возвращается информация о пользователе, включая имя пользователя (`username`) и электронную почту (`email`).
2. `BotCreateView` - класс представления для создания объекта модели `Bot` с использованием сериализатора `BotDetailSerializer`. При создании бота, происходит получение пользователя из запроса и добавление его в данные сериализатора для сохранения объекта.
3. `BotsListView` - класс представления для получения списка объектов модели `Bot`, отфильтрованных по `login\_id`, который извлекается из запроса с помощью функции `get\_user\_id\_from\_request`.
4. `TypeCommandListView` и `CommandsListView` - классы представлений для получения списка объектов моделей `TypeCommand` и `Command` соответственно. Используются разрешения `IsAuthenticated`.
5. `CommandCreateView` - класс представления для создания объекта модели `Command` с использованием сериализатора `CommandDetailSerializer`. Также требует аутентификации пользователя.
6. `CallsListView` и `CallsCreateView` - классы представлений для получения списка объектов модели `CommandCall` и создания нового объекта соответственно. Требуется аутентификация пользователя.
7. `MediaCreateView` и `MediaListView` - классы представлений для создания и получения списка объектов модели `Media`. Требуется аутентификация пользователя.
8. `MessageCommandCreateView` и `MessageCommandListView` - классы представлений для создания и получения списка объектов модели `MessageCommand`. Требуется аутентификация пользователя.
9. `MailCommandCreateView` и `MailCommandListView` - классы представлений для создания и получения списка объектов модели `MailCommand`. Требуется аутентификация пользователя.
10. `BotChatCreateView` и `BotChatListView` - классы представлений для создания и получения списка объектов модели `BotChat`. Требуется аутентификация пользователя.
11. `LinkCreateView` и `LinkListView` - классы представлений для создания и получения списка объектов модели `LinkCommand`. Требуется аутентификация пользователя.
12. `CommandDetailView`, `CommandTypeDetailView`, `CallsDetailView`, `MediaDetailView`, `MessageCommandDetailView`, `MailCommandDetailView`, `BotChatDetailView`, и `LinkDetailView` - классы представлений для получения, обновления и удаления конкретных объектов соответствующих моделей.
13. `BotDetailView` - класс представления для получения, обновления и удаления конкретного объекта модели
14. `Bot`. При получении GET-запроса на данный эндпоинт, возвращается информация о боте. При отправке DELETE-запроса, бот удаляется. При отправке PUT- или PATCH-запроса, происходит обновление данных бота. Для доступа к данному представлению также требуется аутентификация пользователя.

Каждое представление определяет требуемые разрешения доступа (permissions) и сериализаторы (serializers) для работы с соответствующими моделями. Они используются в Django REST Framework для обработки запросов к API и предоставления данных в формате JSON.

### 4.6 Описание миграций

Миграции в Django представляют собой автоматически генерируемый и выполняемый код, который изменяет схему базы данных. Они позволяют разработчикам легко изменять структуру базы данных без необходимости вручную создавать или изменять таблицы, столбцы и отношения.

Так образом все описаные модели в файле models.py, которые были описаны ранее будут созданы записи в базе данных после применения миграции, а так же добавится запись о суперпользователе.

## Разработка клиентской составляющей веб-приложения

Клиентская часть веб-приложения на React JS является фронтенд-частью приложения, которая отвечает за пользовательский интерфейс и взаимодействие пользователя с приложением.

React JS - это JavaScript-библиотека, которая позволяет создавать пользовательские интерфейсы с использованием компонентного подхода. Она позволяет разделить пользовательский интерфейс на множество маленьких, независимых компонентов, которые могут быть повторно использованы и легко модифицированы.

Для начала разработки клиентской части приложения на React JS необходимо настроить окружение, установить необходимые зависимости и настроить сборку проекта. После этого можно приступить к созданию компонентов, которые будут составлять интерфейс приложения.

Компоненты в React JS могут быть функциональными или классовыми. В функциональных компонентах используется функция, возвращающая JSX (расширение синтаксиса JavaScript, которое позволяет описывать структуру пользовательского интерфейса). В классовых компонентах используется класс, который наследуется от базового класса React.Component и имеет метод render(), в котором возвращается JSX.

Компоненты могут содержать внутри себя другие компоненты, что позволяет создавать иерархию компонентов и компонентный подход позволяет удобно управлять состоянием приложения и обновлять интерфейс при изменении данных.

Клиентская часть приложения на React JS может включать в себя следующие основные элементы:

1. **Роутинг**: Можно использовать библиотеку React Router для настройки маршрутизации в приложении. Это позволяет определить различные пути URL и связать их с соответствующими компонентами, чтобы реализовать навигацию между разными страницами.
2. **Компоненты форм**: Для работы с формами можно использовать различные компоненты, такие как <input>, <select>, <textarea> и другие. Компоненты форм могут иметь своё собственное состояние и обрабатывать пользовательский ввод.
3. **Компоненты списка**: Если требуется отображение списка элементов, можно использовать компоненты списка, такие как <ul> и <li>. Компонент списка может принимать данные из внешнего источника, например, из базы данных или API, и отображать элементы списка на основе этих данных.
4. **Взаимодействие с API**: Чтобы получать и отправлять данные на сервер, можно использовать сетевые запросы с помощью библиотеки Axios или встроенного API Fetch. Это позволяет обновлять данные в приложении, обмениваться информацией с сервером и поддерживать актуальность данных.
5. **Стилизация**: Для стилизации компонентов можно использовать CSS или CSS-препроцессоры, такие как SASS или LESS. Также можно использовать библиотеки стилей, такие как Bootstrap или Material-UI, которые предоставляют готовые компоненты и стили для быстрой разработки интерфейса.
6. **Обработка событий**: В React JS можно добавлять обработчики событий к компонентам, например, для обработки кликов, изменения состояния или отправки данных на сервер. Это позволяет реагировать на действия пользователя и обновлять интерфейс соответствующим образом.

Компоненты, используемые на странице, могут быть повторно использованы в других частях приложения. Это позволяет создавать модульную и масштабируемую клиентскую часть. Важно обеспечивать низкую связность между компонентами, чтобы они были независимы друг от друга. Это повышает возможность расширения функциональности и надежности всего приложения.

Клиентская часть веб-приложения на React JS предоставляет пользователю удобный и отзывчивый интерфейс, который взаимодействует с серверной частью приложения для обработки данных и выполнения задач.

## Реализация программы для поддержки ботов

Система должна обеспечивать автоматическую генерацию и запуск ботов в Telegram, соответствующих функционалу и сценариям поведения, заданным в базе данных. Процесс заполнения БД осуществляется с использованием веб-приложения.

Используя токен входных данных и под управлением требований Telegram API и требований к безопасности, система управления ботами создает новый объект бота с помощью Telegram API на сервере. Этот процесс включает назначение основных параметров и установку начального состояния бота.

Полученный объект бота и его параметры затем передаются на этап настройки функциональности. Система управления ботами, работающая на сервере, устанавливает необходимые функции бота в соответствии с требованиями к безопасности и функционалу. Это может включать настройку обработки входящих сообщений, автоматические ответы, кнопки встроенной клавиатуры и другие функции.

Под управлением требований к функционалу и производительности, система управления ботами на сервере принимает параметры бота и токен для установки базовых параметров бота. Эти параметры могут включать настройки профиля бота, начальные команды, параметры подключения. Результатом являются конкретные команды бота и сценарий поведения бота.

Под управлением требований к функционалу и производительности, система управления ботами на сервере устанавливает набор команд бота, которые были определены на предыдущем этапе. Эти команды будут использоваться ботом для обработки входящих сообщений от пользователей.

Система управления ботами устанавливает сценарий поведения бота на сервере в соответствии с требованиями к функционалу и производительности. Это включает в себя определение последовательности действий, которые бот будет выполнять в ответ на определенные события или команды от пользователей.

Используя установленные команды и сценарии, система управления ботами подключается к API Telegram в соответствии с требованиями к безопасности и требованиями Telegram API. В результате получаем настроенный бот, готовый к взаимодействию с пользователями через Telegram API.

Система управления ботами на сервере используется для обновления данных о боте в базе данных. Механизмы, которые задействованы в этом подпроцессе, включают сервер, базу данных и систему управления ботами. Результатом являются измененные данные, которые теперь содержат актуальную информацию о настройках бота.

Токен подаются на вход этого подпроцесса. С помощью сервера, Telegram API и системы управления ботами запускается обработчик входящих сообщений. Этот процесс происходит под управлением требований к функционалу и производительности. Результатом является запущенный обработчик, который теперь готов обрабатывать входящие сообщения от пользователей.

После того как обработчик входящих сообщений был запущен, бот переводится в режим ожидания. Это означает, что бот теперь готов к работе и ожидает входящих сообщений от пользователей. На вход этого подпроцесса подаются токен и запущенный обработчик. В этом процессе используются такие механизмы, как сервер, Telegram API и система управления ботами. Все это происходит под управлением требований к функционалу и производительности. Результатом является созданный бот, готовый к работе, и обновленная информация о боте в базе данных.

## Руководство пользователя

Главная страница приложения представлена на рисунке 7.1.

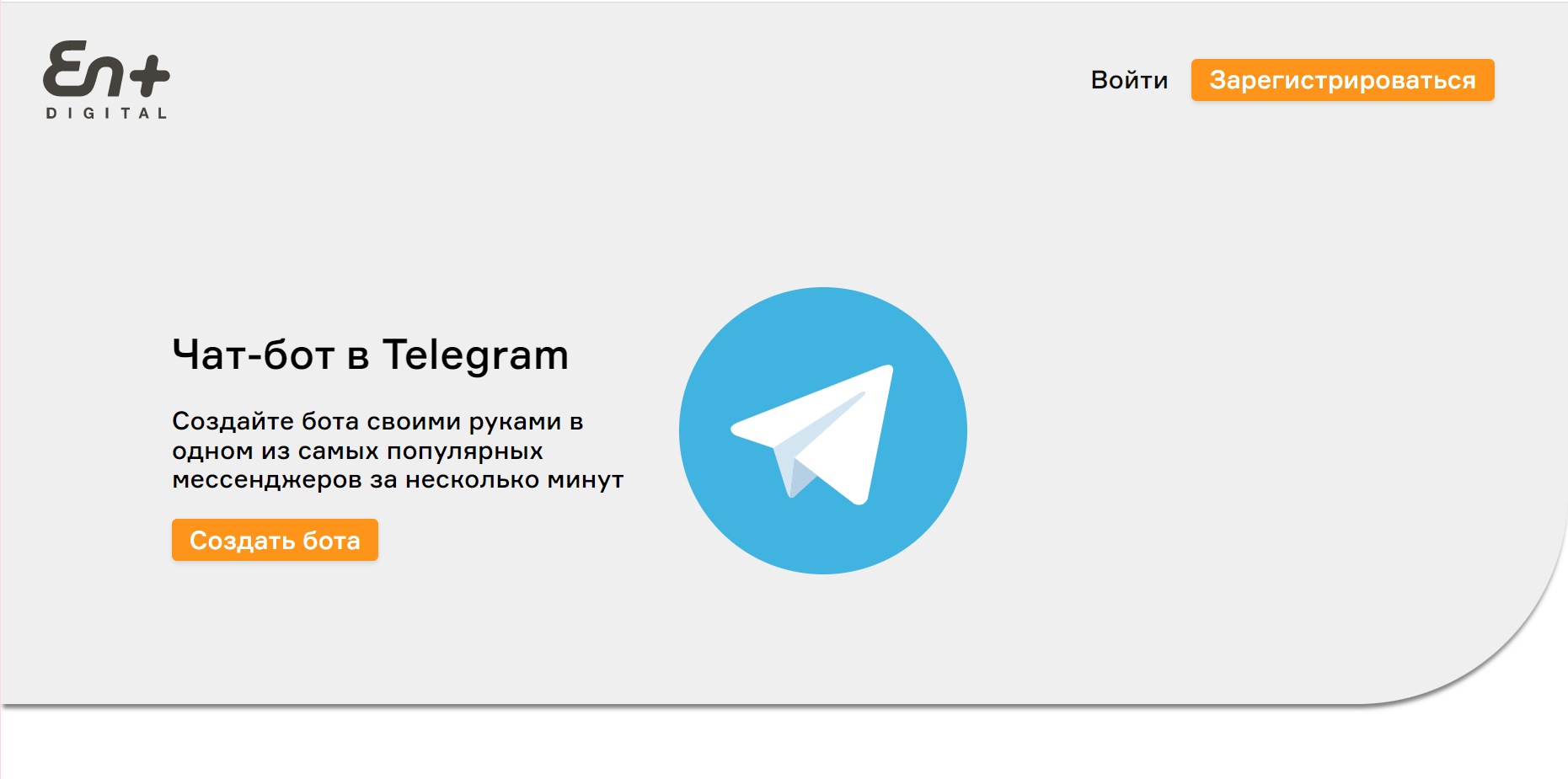


Рисунок 7.1 – Главная страница приложения

### Авторизация

Для использования приложения пользователь должен авторизоваться в системе, для этого необходимо, при отсутствии аккаунта, зарегистрироваться предоставив электронную почту, придумав пароль и имя пользователя (см. рисунок 7.1.1).

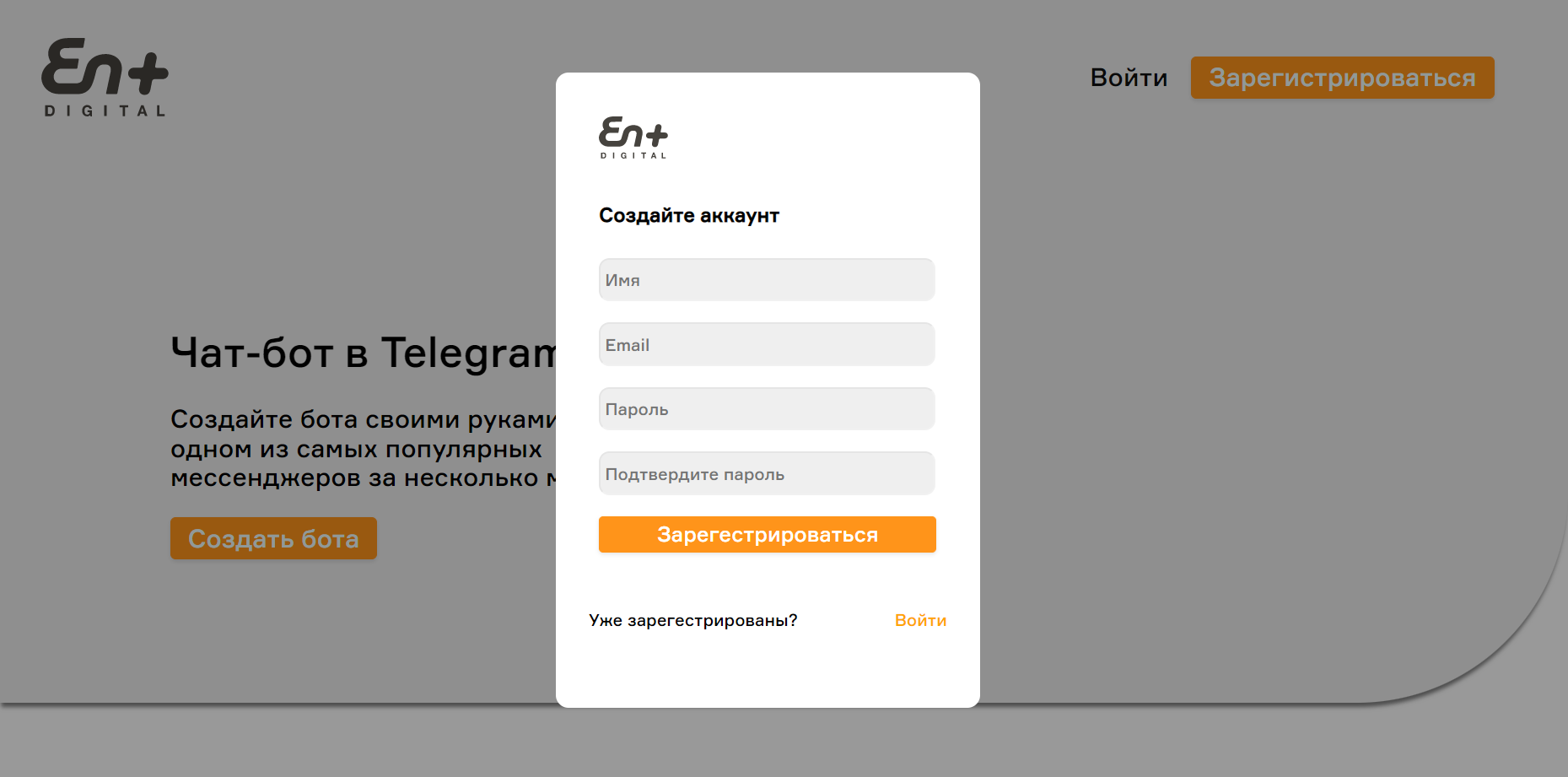


Рисунок 7.1.1 – Форма регистрации

При наличии аккаунта пользователь должен в него войти использую имя пользователя и пароль (см. рисунок 7.1.2).

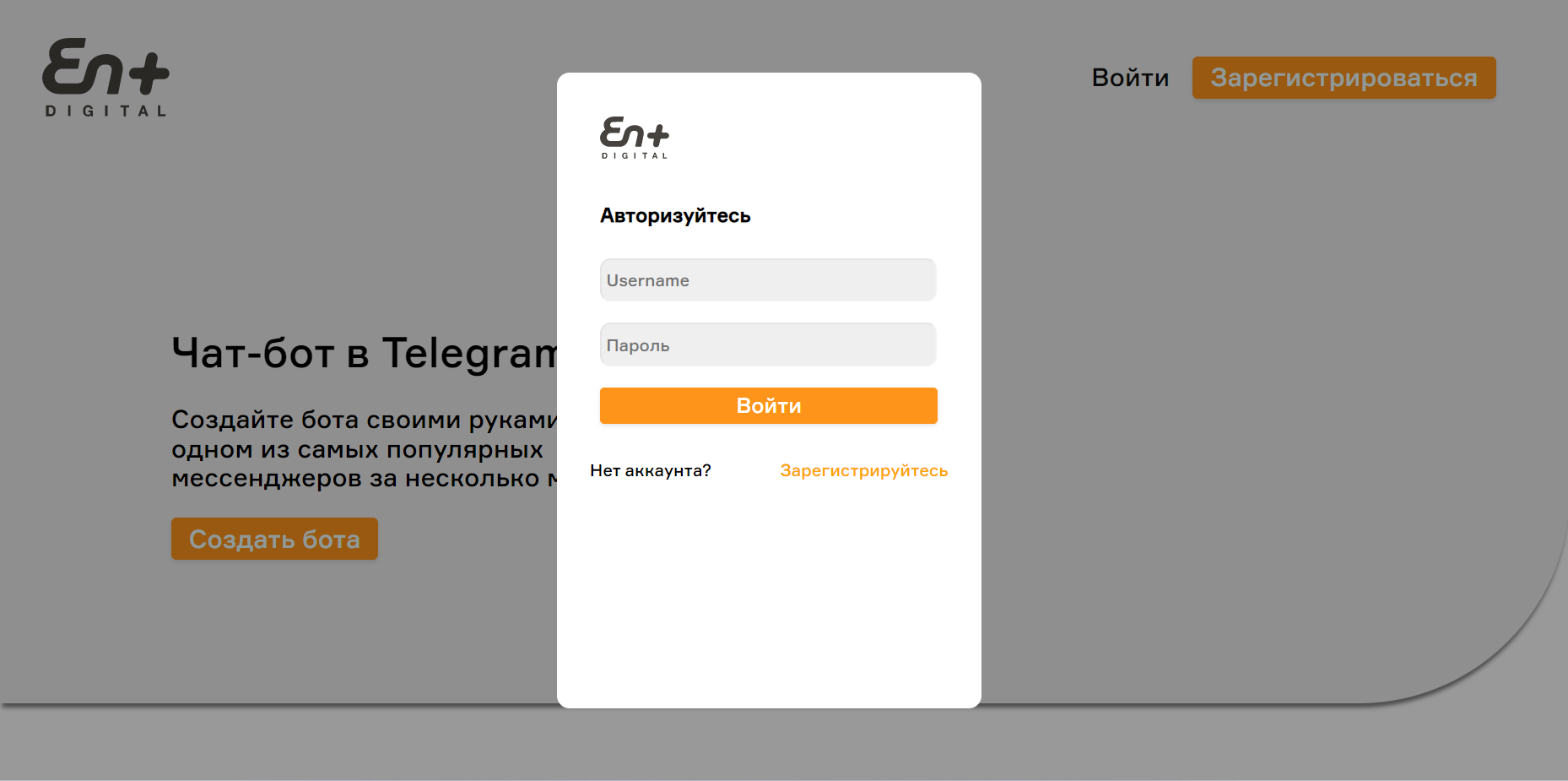


Рисунок 7.1.2 – Форма авторизации

### Список ботов

В окне «Список ботов» (рисунок 7.2.1) пользователь может создать бота, заполнив поле «Имя бота» и нажав на соответствующую кнопку «Создать бота» (при пустом поле имени бота высветится предупреждение).

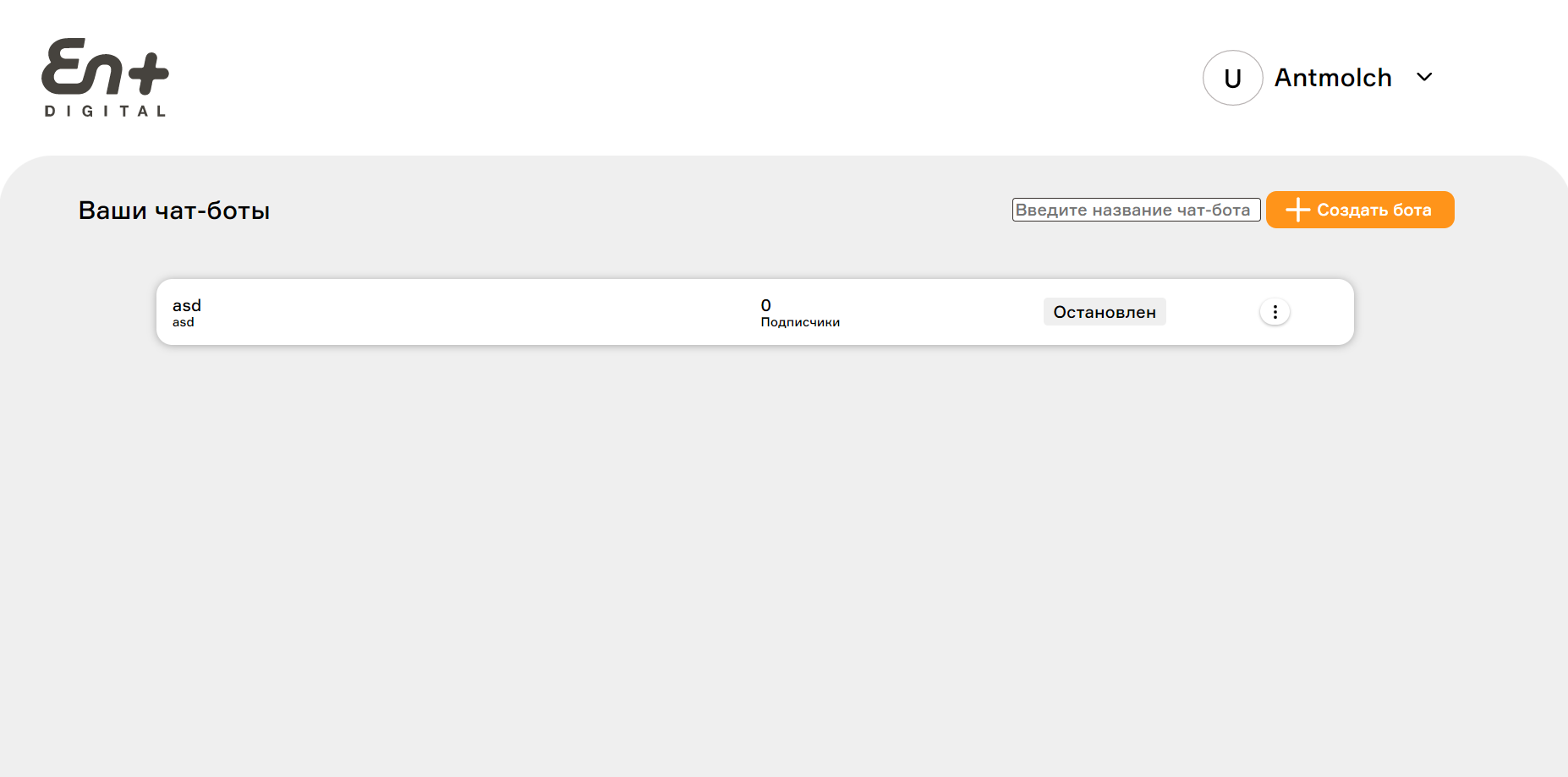


Рисунок 7.2.1 – Список ботов

После чего пользователь попадает в меню руководства по созданию бота (см. рисунок 7.2.2). После выполнения всех пунктов в списке ботов появится созданный бот.

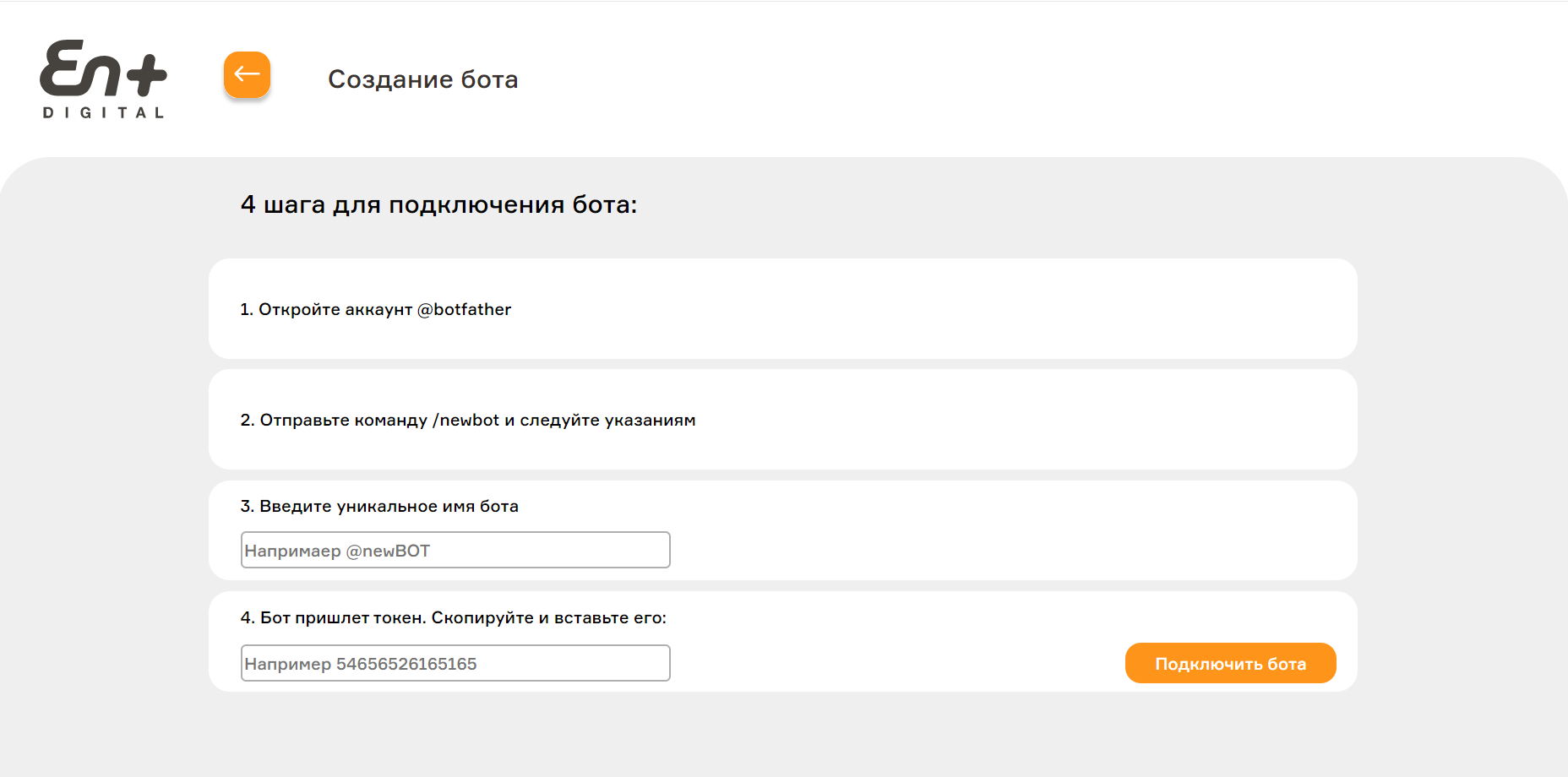


Рисунок 7.2.2 – Меню руководства по созданию бота

Нажав на созданном боте кнопку действий с ботом откроется меню возможных операций (см. рисунок 7.2.3).

**Описание операций:**

* **Изменить** – откроется конструктор бота;
* **Подключить / Отключить** – произведёт включение или отключение бота;
* **Удалить** – безвозвратно удалит бота.

Также по нажатию на поле с название бота откроется его конструктор.

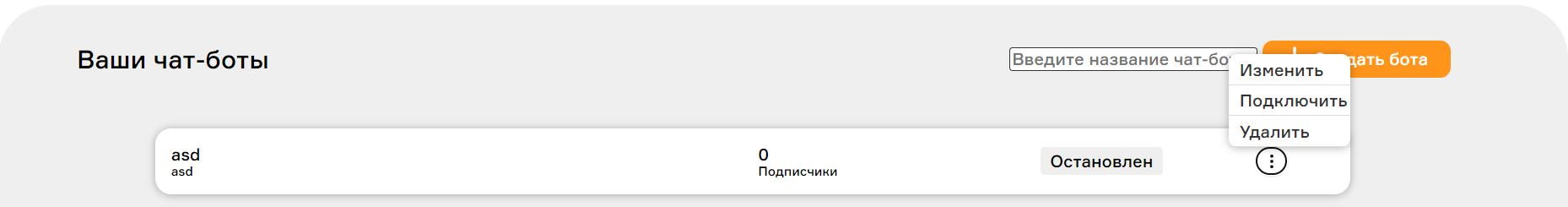


Рисунок 7.2.3 – Меню бота

В заголовке сайта при нажатии на стрелочку справа от имени пользователя откроется меню с возможностью изменения пароля и выходом из аккаунта (см. рисунок 7.2.4).

При выборе пункта изменение пароля на электронную почту пользователя придёт сообщение с ссылкой на страницу изменения пароля.

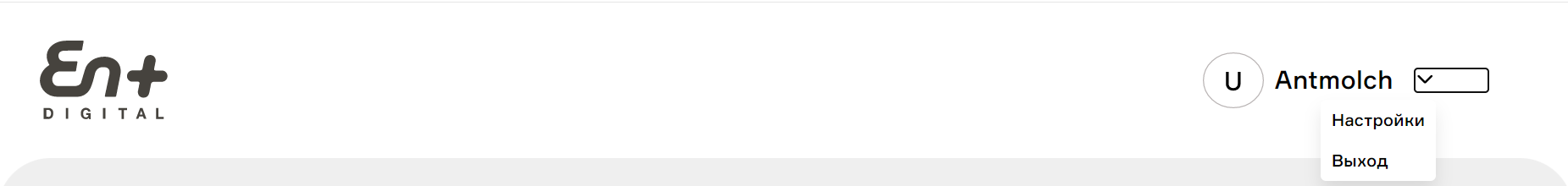


Рисунок 7.2.4 – Меню аккаунта

### Конструктор

Окно конструктора чат-бота представлено на рисунке 7.3.1.

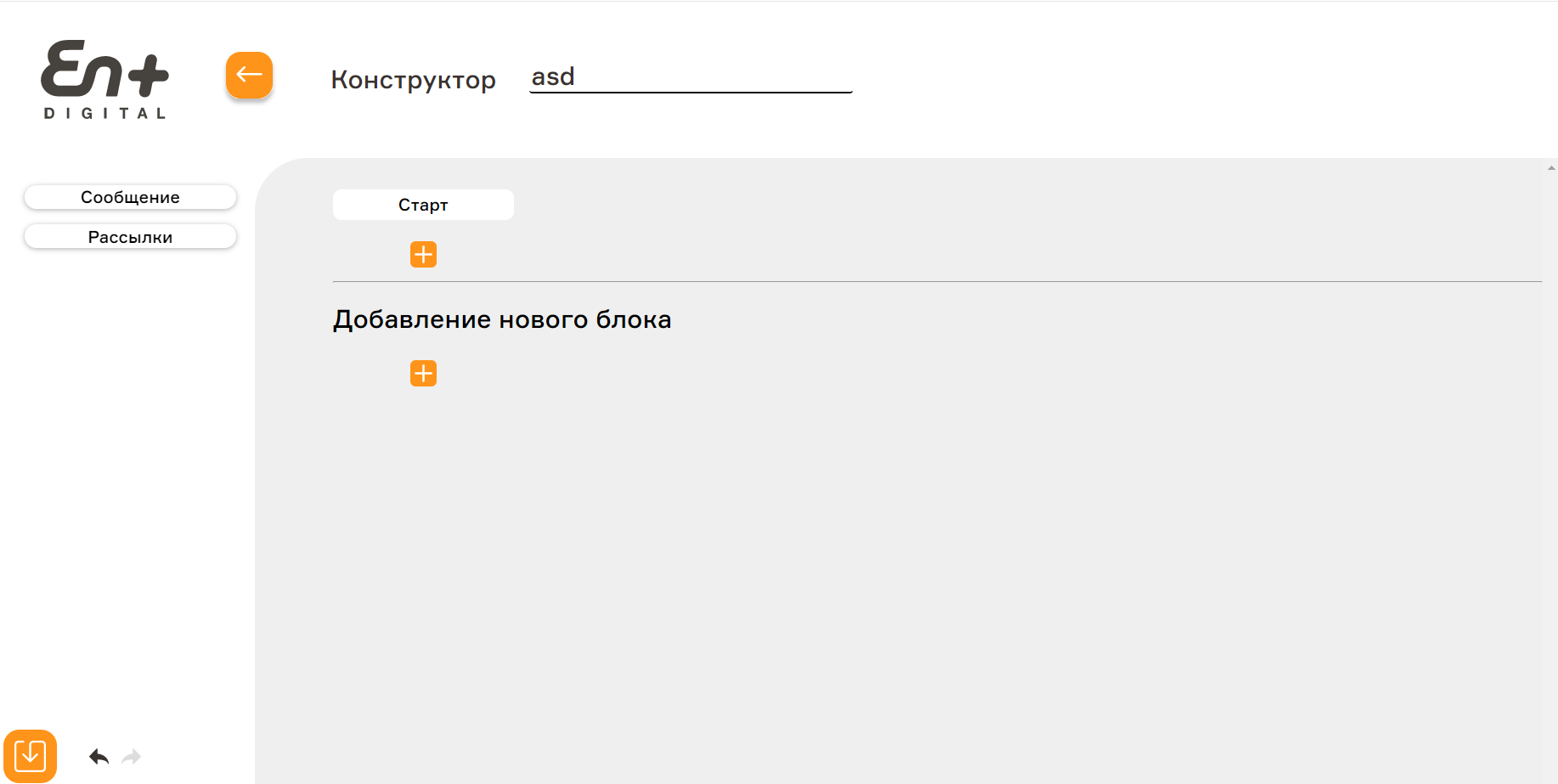


Рисунок 7.3.1 – Окно конструктора

Оно состоит из трёх компонентов: заголовок (шапка сайта), меню команд (левая часть окна) и самого конструктора (центральной серой области).

В компоненте заголовка (см. рисунок 8.3.2). можно вернуться на предыдущую страницу и изменить имя бота



Рисунок 7.3.2 – Компонент заголовка

В компоненте меню команд (см. рисунок 7.3.3). предоставлен выбор из двух типов команд (сообщение и рассылка). Также в нижней части компонента есть кнопки для сохранения бота и отката действий (максимальное количество отмен действий – 10).



Рисунок 7.3.3 – Компонент меню команд

В компоненте конструктора (см. рисунок 7.3.4) представлена структура бота.

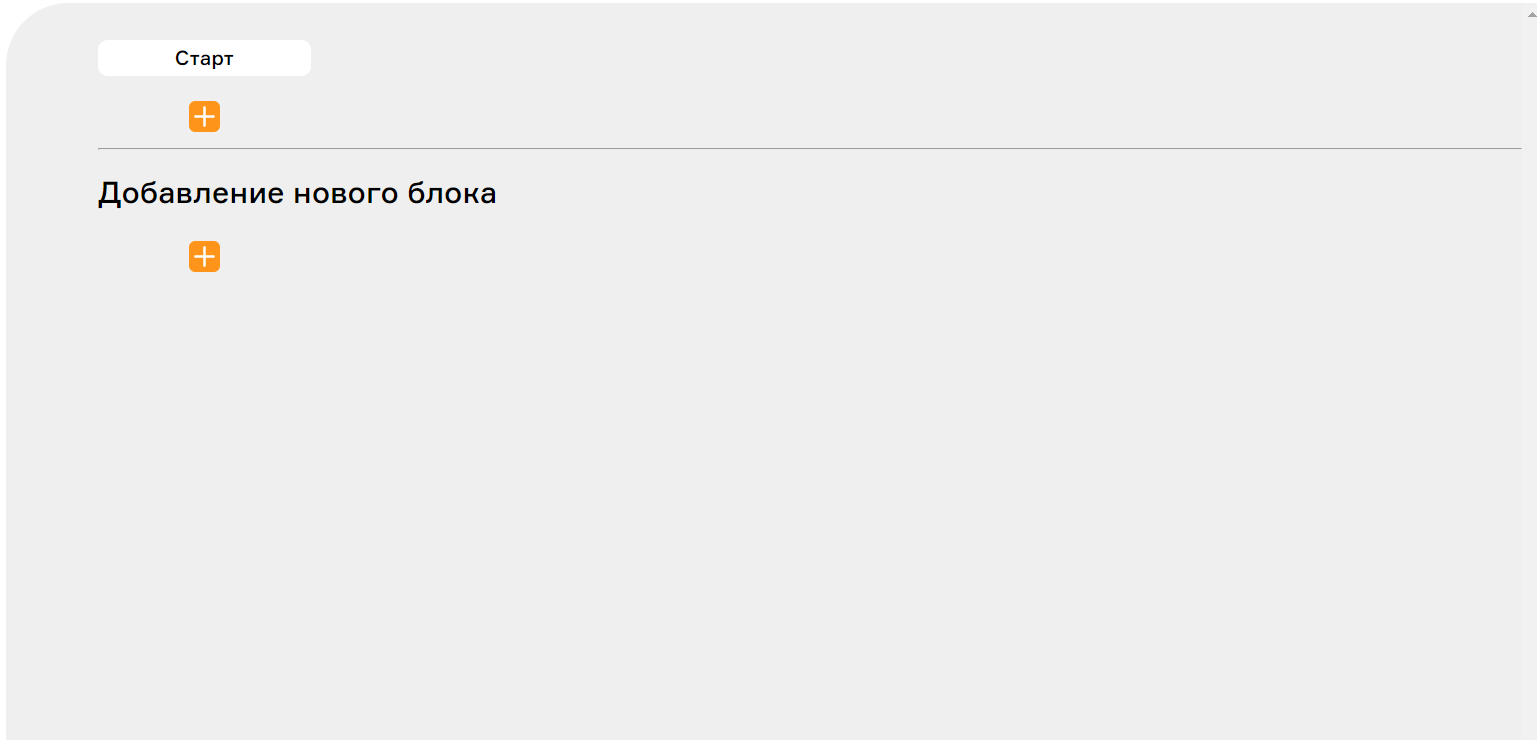


Рисунок 7.3.4 – Компонент конструктора

При выборе типа команды и нажатии на место для добавления (помечено знаком «+»). Произойдёт добавление команды.

**Правила для добавления команд:**

1. Команда рассылки не может быть стартовой командой и не может идти после команды сообщения. Для её добавления необходимо выбрать место под надписью: «Добавление нового блока». После чего она появится в нижней части конструктора с соответствующей подписью (см. рисунок 7.3.5).
2. Команда сообщения может быть добавлена в любое отмеченное место (см. рисунок 7.3.6).

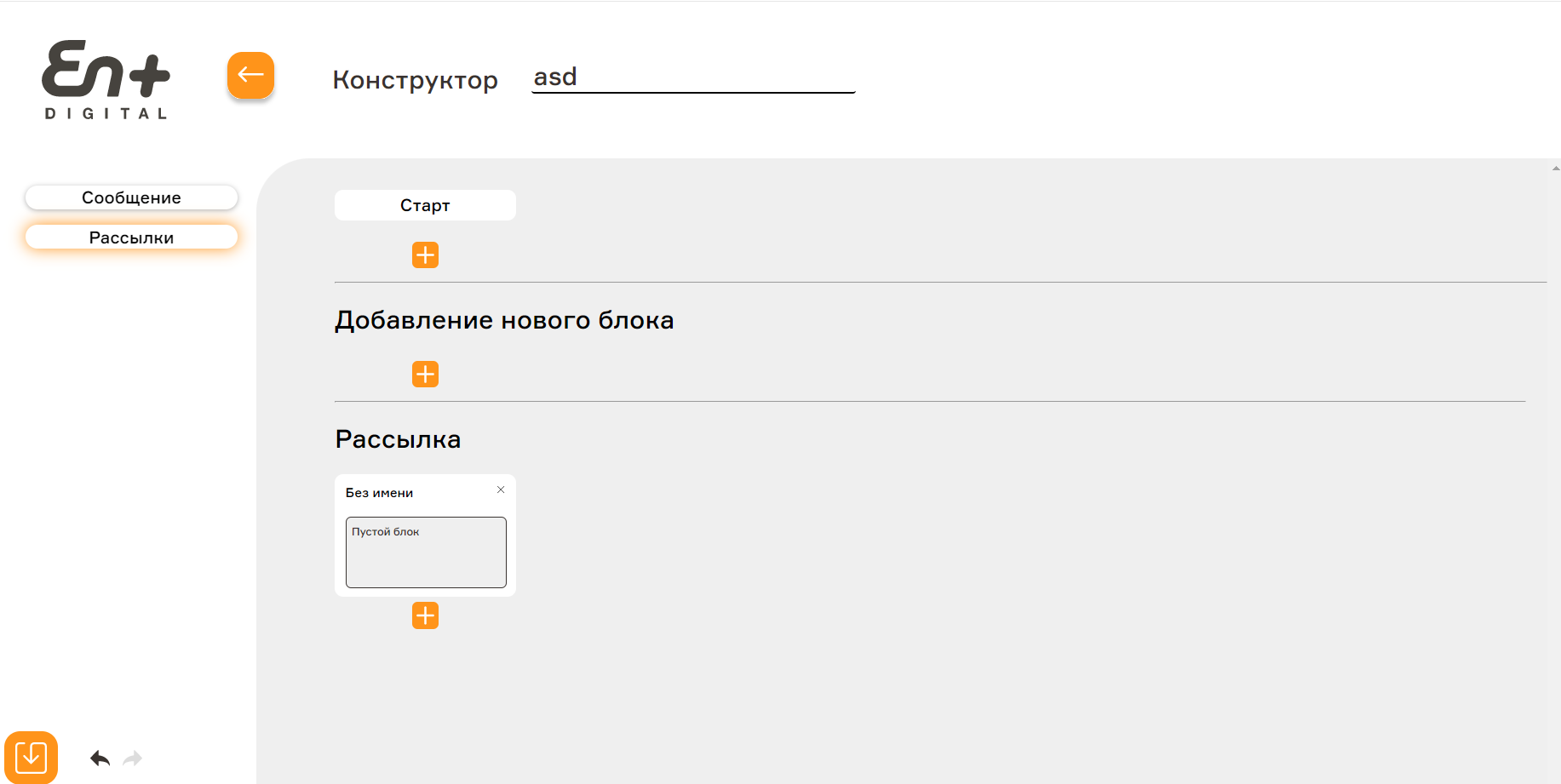


Рисунок 7.3.5 – Добавление команды «Рассылка»

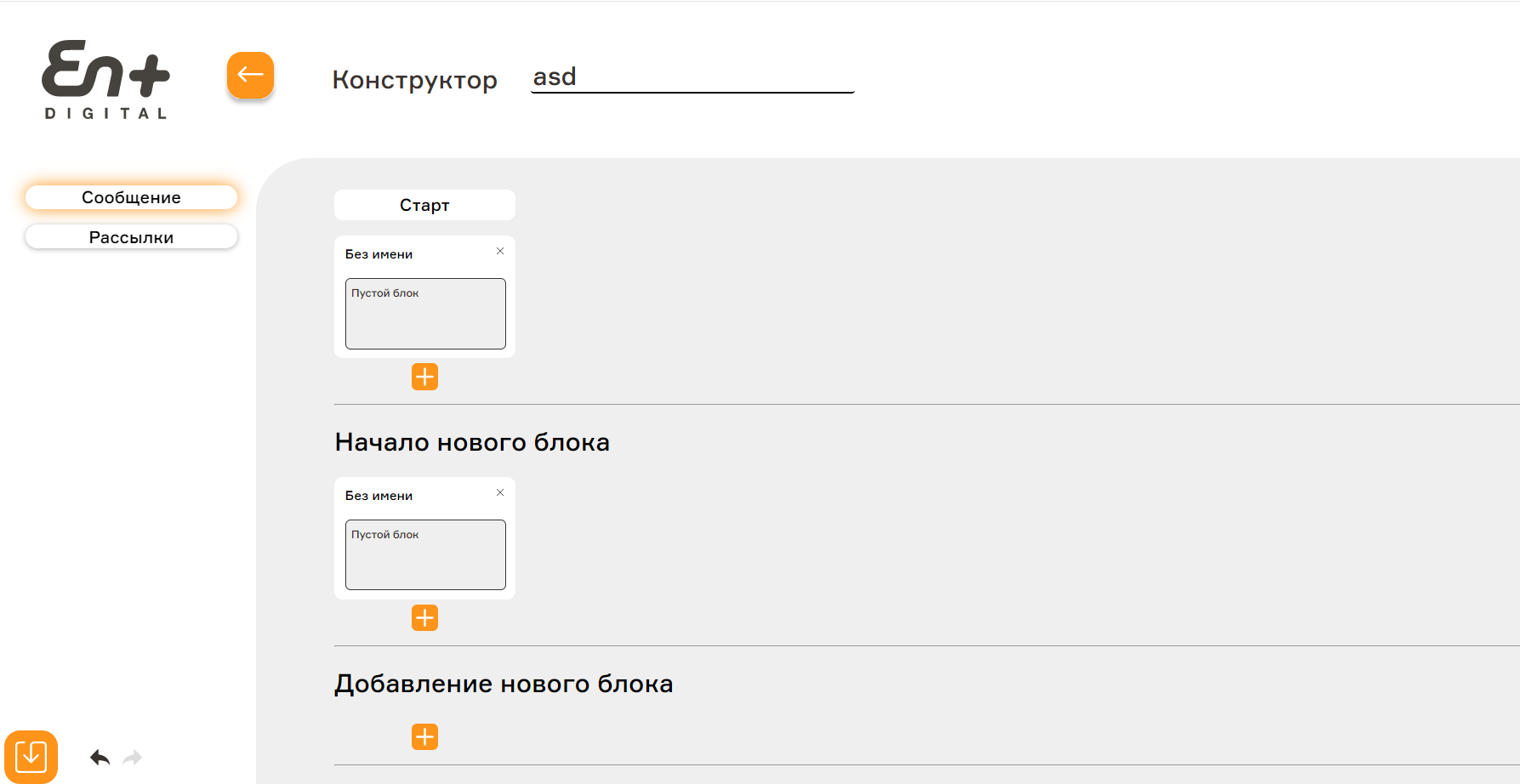


Рисунок 7.3.6 – Добавление команд «Сообщение»

Пример созданной структуры бота представлен на рисунке 7.3.7

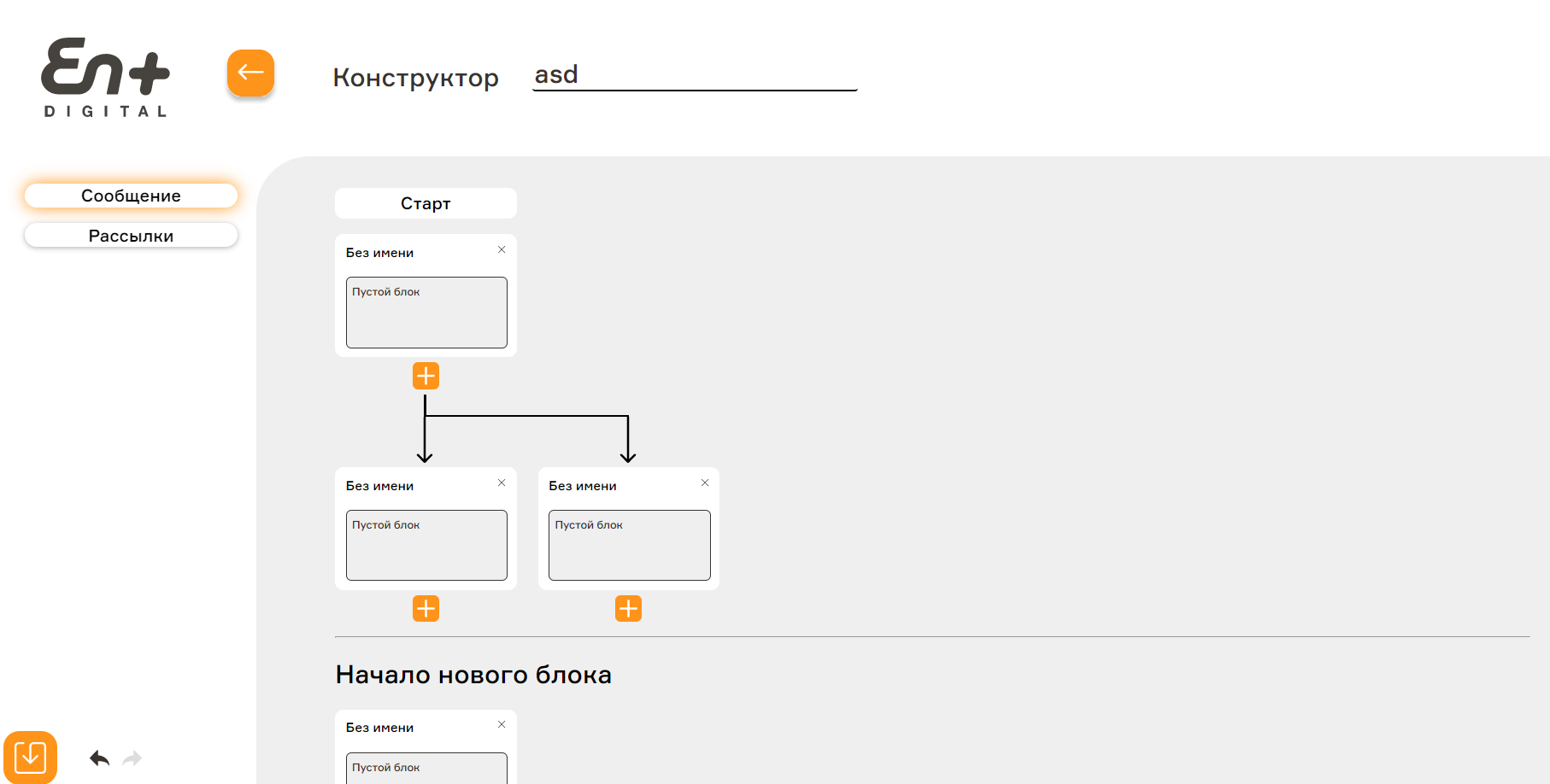


Рисунок 7.3.7 – Пример созданной структуры бота

Для удаления команды нужно нажать на крестик в её правом верхнем углу.

Для изменения команды нужно нажать на неё, после чего всплывёт окно для её изменения. Окно для изменения команды «Сообщение» представлен на рисунке 7.3.8, а для команды «Рассылка» на рисунке 7.3.9.

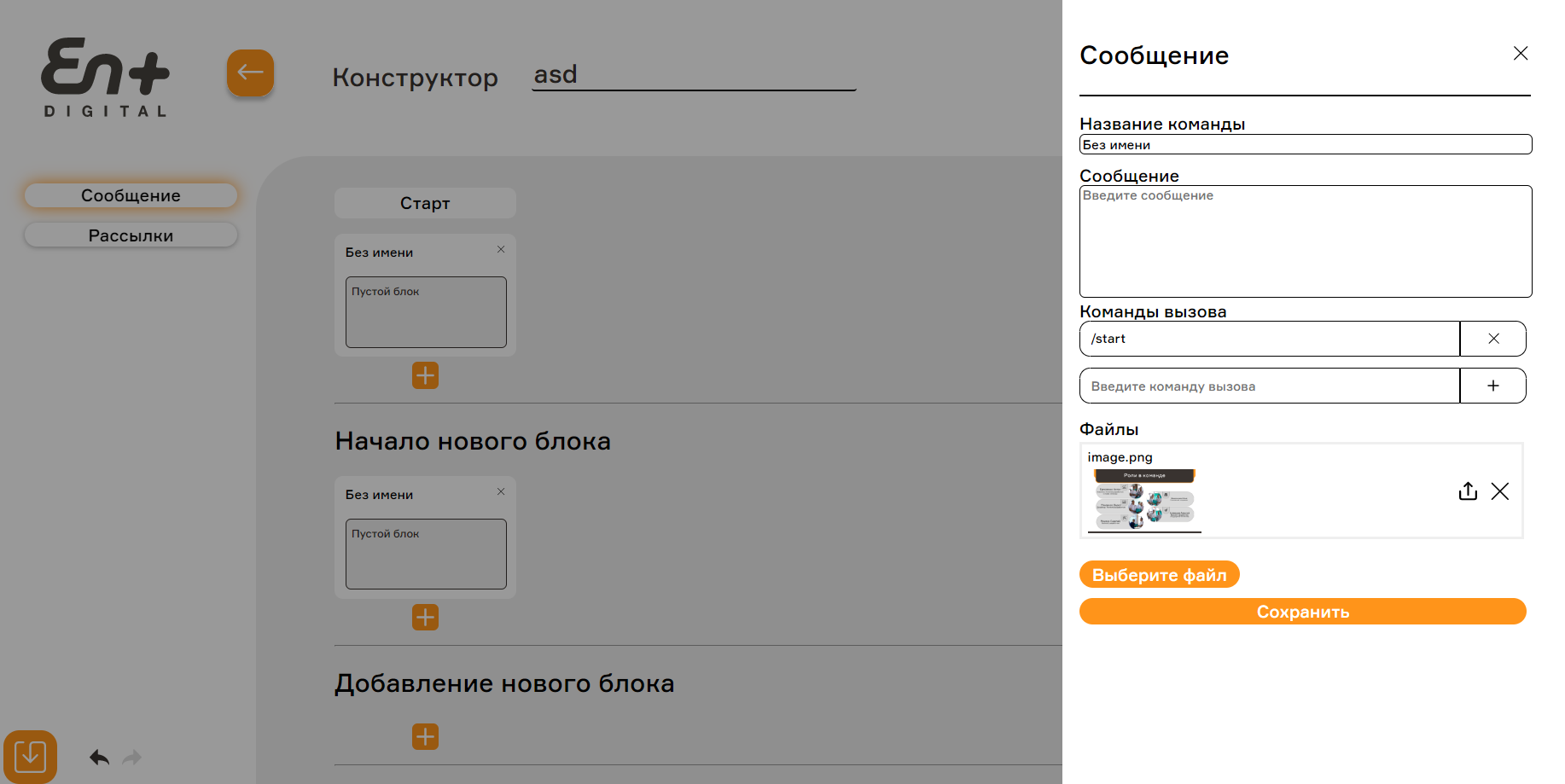


Рисунок 7.3.8 – Окно изменения команды «Сообщение»

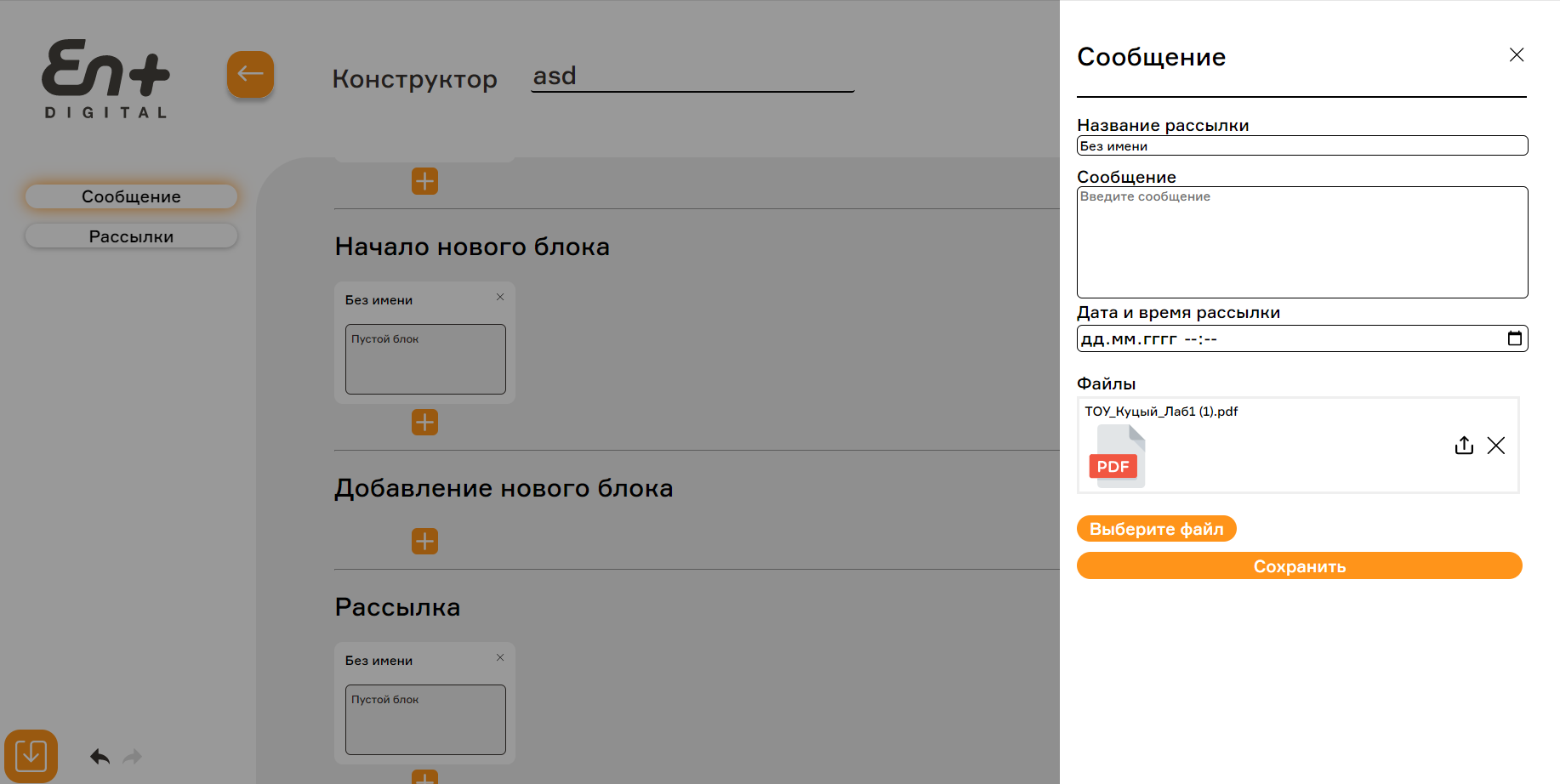


Рисунок 7.3.9 – Окно изменения команды «Рассылка»

Для команды «Сообщение» предлагается изменить её имя, текст сообщения, вызываемые фразы и прикрепить файл формата pdf, docx, mp4, jpeg, png.

Для команды «Рассылка» предлагается изменить её имя, текст сообщения, дату и время вызова и прикрепить файл формата pdf, docx, mp4, jpeg, png.

Ограничения по добавлению файлов – 1 файл не больше 4Мб на команду.

**Для добавления фраз вызова есть ограничения:**

* Вызов не должен совпадать с вызовом другой команды;
* Вызов не должен повторяться.

Для сохранения изменений в команде необходимо нажать на соответствующую кнопку «Сохранить».

Для выхода из окна без сохранения необходимо нажать на крестик ввернем правом углу, либо вне границ окна.

### Проверка работы бота

После создания и подключения бота следует перейти в Telegram и найти своего бота по его уникальному имени (см. рисунок 7.4.1).

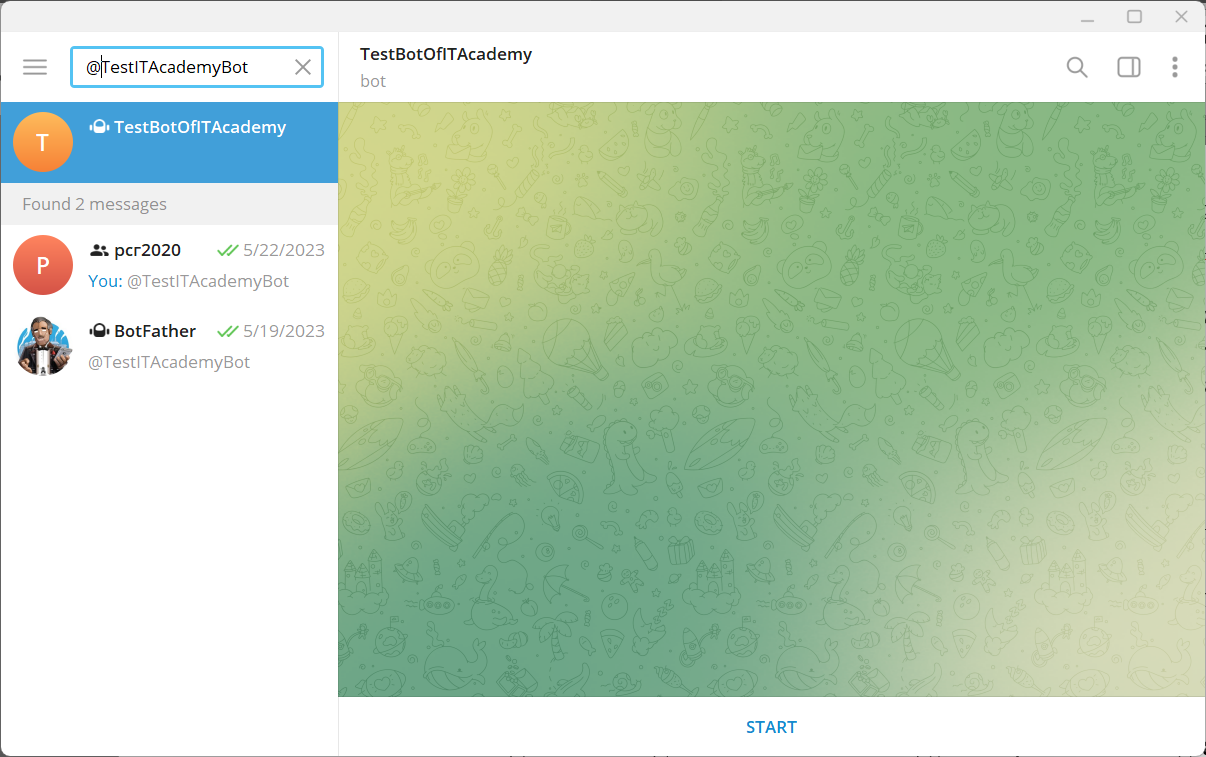


Рисунок 7.4.1 ­ Поиск бота

На рисунке 7.4.2 представлен пример работы Telegram бота, созданного при помощи конструктора.



Рисунок 7.4.2 ­Пример работы бота

## Перспективы

В будущем планируется расширять функциональную часть приложения, а именно:

1. Доработать визуальную часть, чтобы работа с приложением стала полностью интуитивно понятной пользователю.
2. Улучшить масштабируемость.
3. Переработать связь с сервером, для улучшения производительности.
4. Добавить возможность создания ботов для других мессенджеров, например, для Viber.
5. Добавить подробную статистику бота.
6. Добавить новые типы команд.
7. Реализовать нейронную сеть, которая будет определять то, что нужно пользователю по его сообщению.

Веб-приложения "Конструктор чат-ботов" имеет потенциал для монетизации. Можно рассмотреть различные модели монетизации, такие как платные подписки, продажа расширенных функций или услуг, интеграция с рекламными платформами и другие. Успешное коммерческое использование приложения может принести значительную прибыль и стать основой для развития и расширения проекта.

# Заключение

Веб-приложение «Конструктор чат-ботов» предлагает гибкий и адаптивный подход к созданию чат-ботов. Это означает, что пользователи могут создавать и настраивать чат-ботов с учетом своих уникальных потребностей и требований. Благодаря гибкости приложения, оно может быть использовано в различных отраслях и сферах деятельности.

В ходе работы были изучены и применены различные технологии и инструменты для создания веб-приложения, способного конструировать и настраивать чат-ботов. Анализ предметной области позволил определить основные требования к приложению и его функциональность.

В ходе разработки приложения были приобретены ценные навыки в области web-программирования, взаимодействия с API, создания пользовательских интерфейсов и разработки серверной части приложения. Работа над проектом позволила практически применить полученные знания и получить опыт разработки полноценного веб-приложения.

# Список использованных источников

1. Документация на фреймворк Django – <https://docs.djangoproject.com/en/4.2/> (дата обращения 15.05.2023).
2. Документация на React JS – <https://legacy.reactjs.org/docs/getting-started.html> (дата обращения 10.05.2023).
3. Web-дизайн: удобство использования Web-сайтов / Якоб Нильсен, Хоа Лоранжер ; [пер. с англ. и ред. В. С. Иващенко]. - Москва [и др.] : Вильямс, 2007. - 366 с. : ил.; 24 см.; ISBN 978-5-8459-1222-0.
4. Информация о работе с css – <https://developer.mozilla.org/ru/> (дата обращения 12.05.2023).
5. Документация на C# – <https://learn.microsoft.com/ru-ru/dotnet/csharp/> (дата обращения 15.05.2023).
6. Документация на Telegram Bot API – <https://core.telegram.org/bots/api> (дата обращения 25.04.2023).