

Министерство просвещения РФ
Федеральное государственное бюджетное образовательное
учреждение высшего образования
«Уральский государственный педагогический университет»
Институт математики, физики, информатики и технологий
Кафедра информатики, информационных технологий
и методики обучения информатике

ПРИЛОЖЕНИЕ-КОНСТРУКТОР ЧАТ-БОТОВ «EDUCATION-BOT-CREATOR»

*Выпускная квалификационная работа
бакалавра по направлению подготовки
09.03.02 – Информационные системы и технологии*

Работа допущена к защите
«___» _____ 2022 г.
Зав. кафедрой _____

Исполнитель: студент группы ИСиТ 1701z
Института математики, физики,
информатики и технологий
Мурашов А.А.

Руководитель: кандидат педагогических наук,
доцент кафедры ИИТ и МОИ
Арбузов С.С.

Екатеринбург – 2022

Реферат

Тема выпускной квалификационной работы: «Приложение-конструктор чат-ботов «Education-bot-creator».

Выпускная квалификационная работа выполнена на 100 страницах, содержит 5 таблиц, 57 рисунков, 54 использованных источника.

Предмет разработки – web-приложение для создания образовательных чат-ботов.

Цель работы – спроектировать и разработать web-приложение для создания образовательных чат-ботов.

Web-приложение реализовано с помощью языка программирования TypeScript и JavaScript. Используемая база данных – MongoDB.

В качестве сопроводительной документации создано руководство пользователя.

Оглавление

| | |
|--|----|
| Реферат | 2 |
| Введение..... | 4 |
| Глава 1. Теоретические основы создания и использования чат-ботов | 6 |
| 1.1. Технологии создания web-приложений для разработки чат-ботов | 6 |
| 1.2. Инструментальные средства разработки web-приложений для создания чат-ботов..... | 16 |
| 1.3. Формализованное описание технического задания..... | 25 |
| Глава 2. Разработка web-приложения «Education-bot-creator»..... | 30 |
| 2.1. Проектная часть web-приложения «Education-bot-creator» | 30 |
| 2.2. Разработка основных элементов web-приложения «Education-bot-creator» | 41 |
| 2.3. Пользовательский интерфейс web-приложения «Education-bot-creator» | 46 |
| 2.4. Тестирование web-приложения «Education-bot-creator» | 66 |
| Заключение | 69 |
| Список информационных источников..... | 70 |
| Приложения | 75 |

Введение

В настоящее время Интернет выступает как одно из самых популярных средств массовой информации, вытесняя телевидение, радио и печатные издания. Создание web-приложений – является значительным инструментом, необходимым для решения маркетинговых стратегий.

Трудно себе представить компанию, у которой нет своего сайта. Таким приложением может быть, как сайт-визитка, так и интернет-приложение. Компании используют ресурсы сети Интернет в рекламных целях, а также как непосредственная площадка для осуществления коммерческой деятельности.

Использование сети Интернет дает ряд преимуществ, это сравнительно небольшие затраты на регистрацию, возможность клиенту получить более полную информацию и многое другое.

Также в наши дни нашло свое отражение практика использования чат-ботов. Они используются практически во всех сферах: от электронной коммерции до промышленности и сферы образования. Особенностью системы образования является необходимость работы педагогов, относящихся к разным поколениям, с детьми или молодыми людьми, являющихся представителями иного, последующего поколения, обладающего принципиально иными характеристиками, навыками и жизненными принципами.

В эпоху всеобщей цифровизации создаются новые технологии и сервисы, которые можно эффективно использовать в образовательном процессе и которые будут интересны представителям молодых поколений. В последние несколько лет актуальной тенденцией в IT-индустрии стало создание чат-ботов, которые имеют настолько большой потенциал в использовании, что, как считают эксперты, в будущем заменят собой множество приложений, интернет-поисковиков и даже приведут к исчезновению некоторых профессий.

Одним из основных факторов, определивших активное создание и успешное использование чат-ботов, является повсеместное распространение

мессенджеров – сервисов быстрых сообщений, и социальных сетей

Актуальность темы определена тем, что в настоящее время значительная доля пользователей предпочитает получать информацию непосредственно из сети Интернет.

Продукт разработки – приложение-конструктор для создания образовательных чат-ботов «Education-bot-creator».

Цель работы – спроектировать и разработать web-приложение для создания образовательных чат-ботов.

Для осуществления основной цели выпускной квалификационной работы были поставлены следующие задачи:

1. Произвести анализ технологий для создания и использования образовательных чат-ботов.
2. Проанализировать и обосновать выбор технологий для разработки web-приложения, позволяющего создавать образовательных чат-ботов.
3. В соответствии с техническим заданием разработать web-приложение «Education-bot-creator».
4. Осуществить тестирование web-приложения «Education-bot-creator» и подготовить сопроводительную документацию по его использованию.

Глава 1. Теоретические основы создания и использования чат-ботов

1.1. Технологии создания web-приложений для разработки чат-ботов

Использование чат-ботов в наши дни нашло отражение практически во всех сферах деятельности: от электронной коммерции до промышленности и сферы образования. В настоящее время значительная доля пользователей предпочитает получать информацию посредством сети Интернет. Чат-боты могут создаваться как с помощью непосредственного написания кода на почти-что любом языке программирования, так и с помощью web-приложений, которые позволяют создать чат-бота используя графический интерфейс.

Перед созданием любого проекта необходимо произвести анализ существующих технологий и способов проектирования и создания web-приложения для разработки чат-ботов.

Под ботом (роботом, интернет-ботом) понимается специальная программа, которая выполняет какие-либо действия через интерфейсы, предназначенные для людей, автоматически, либо согласно некоторому расписанию.

В основном ботов используют для выполнения какой-либо повторяемой работы, чтобы максимизировать скорость ее выполнения. Например, отвечать в службе поддержки какой-либо компании на частые вопросы пользователей, тем самым обеспечивая более быструю реакцию на что-либо по сравнению с человеческой реакцией. Также ботов используют для поиска какой-либо информации в интернете на различных ресурсах, например, прогноз погоды или афиша мероприятий.

Как правило, чат-бот ведет автоматическое общение с пользователем с помощью текста или голоса от лица компании или бренда с целью упрощения онлайн-коммуникации, тем самым предоставляя пользователю актуальную

информацию наиболее оперативно. Таким образом чат-бот используется как альтернативный вариант переписке с живым оператором или звонку менеджеру компании [1].

Чат-бот – это программа, работающая внутри мессенджера (система для обмена мгновенными сообщениями), которая способна отвечать на вопросы, а также самостоятельно их задавать. При этом чат-боты могут выступать еще и в качестве цифровых ассистентов, находясь внутри мессенджера, выполняя различные команды, давая рекомендации и осуществляя некий поиск по параметрам, которые сообщает им пользователь. Чат-бот также может быть частью сайта и, например, быть доступным пользователю через некий виджет [2].

Таким образом, чат-бот – это автоматическая система для общения с пользователями. Другими словами, это алгоритм, робот, который помогает клиентам решать различные задачи. Например, сделать заказ в интернет-магазине, напомнить об акции, принять заявку, помочь сделать заказ, отменить запись к врачу, подтвердить доставку, записать ответы пользователя и т.д. Их не стоит путать с рассылками и информационными каналами. Чат-бот «общается» с пользователем, отвечает на его вопросы. Рассылки же просто дают информацию, ничего не спрашивая и никак не реагируя на запросы читателей. Если рассылку отправляют через чат-бота, то он сможет ответить на вопросы и перейти к диалогу с пользователем [3].

Чат-боты различаются по сложности. Есть простые алгоритмы – они могут вывести информацию по запросу или ответить на вопросы пользователя, если те входят в базу данных. Если ответа в базе нет, робот перенаправляет пользователя к менеджеру. Таких ботов часто собирают самостоятельно с помощью конструкторов.

Чат-бот – это программа, с которой может взаимодействовать не только один пользователь в один момент времени, а гораздо больше, что позволяет охватывать большие аудитории, при этом пользователи могут общаться с

чатботом по-разному: с помощью текстовых сообщений, голоса, жестов и нажатий на элементы интерфейса, если он предусмотрен для конкретного бота. При этом чат-бот доступен пользователям круглосуточно, без перерывов, в отличие от живого оператора [4].

В течение нескольких лет чат-боты, в основном, использовались в средах обслуживания клиентов, но в настоящее время спектр их ролей использования постоянно растет, что обусловлено стремлением к повышению качества обслуживания и эффективности бизнеса на различных предприятиях.

Наиболее полезные чат-боты разрабатываются на основе технологий машинного обучения и отличаются тем, что их обучает человек, и они предназначены для использования людьми, таким образом, человек принимает активное участие на всех стадиях разработки такой программы собеседника. В то время как растет распространение и использование чат-ботов, так же продолжает расти популярность парадигмы взаимодействия «messaging-as-an-interface», согласно которой обмен сообщениями, будучи 7 быстрым, прямым, хорошо визуализированным и позволяющим вести общение сразу с большим количеством собеседников, становится одним из ключевых способов коммуникации. Росту популярности данной парадигмы способствует и то, что большинство предприятий, так или иначе, осваивает социальные сети и мессенджеры в качестве каналов общения с клиентами. Еще одна причина – довольно часто пользователям комфортнее написать свои вопросы и получить прямой ответ, чем задавать их во время «живого» звонка оператору.

Для обеспечения правильной работы чат-бота необходимо помнить о важных методах его взаимодействия с пользователем: машинное обучение или набор правил.

Если алгоритм действий бота основывается на конкретном наборе правил, то его функционал будет являться ограниченным. Бот распознает лишь те команды, которые были изначально указаны при его настройке. Ответ бота формируется только исходя из упомянутого ключевого слова. Если ключевое

слово не распознано, то бот не сможет выдать пользователю верный ответ. Чтобы пользователю было проще общаться и задавать вопросы, лучше создать специальные кнопки, помогающие человеку быстрее определиться с выбором.

Если же алгоритм бота основан на искусственном интеллекте, то при общении с клиентом они поддерживают диалог без упоминания конкретных ключевых слов или фраз. Умный бот самостоятельно совершенствует построение разговора. Таким образом помощник может решать проблемы клиентов еще лучше.

Также для создания чат-ботов могут использоваться web-приложения, которые позволяют создать бота самому, без навыков программирования. Такие приложения, как правило, позволяют создавать цепочки сообщений с пользователем, поддерживают те или иные платформы обмена сообщениями, запрос каких-то данных от пользователя и так далее.

Web-приложение – это клиент-серверное приложение, в котором клиент взаимодействует с веб-сервером при помощи браузера. Логика веб-приложения распределена между сервером и клиентом, хранение данных осуществляется, преимущественно, на сервере, обмен информацией происходит по сети. Одним из преимуществ такого подхода является тот факт, что клиента не зависят от конкретной операционной системы пользователя, поэтому web-приложения являются межплатформенными службами [7].

Основным преимуществом создания web-приложений для поддержки стандартных функций браузера является то, что функции должны выполняться независимо от операционной системы клиента. Вместо написания различных версий для Microsoft Windows, Max OS X, GNU/Linux и других операционных систем, приложение создается один раз для произвольно выбранной платформы и на ней разворачивается. Однако различные реализации CSS, DOM и других спецификаций в браузерах может вызывать проблемы при разработке web-приложений и последующей поддержке. Кроме того, у пользователя есть возможность настраивать многие параметры браузера (например, размер

шрифта, цвета, отключение поддержки сценариев), что может негативно повлиять на корректную работу приложения.

В отличие от вебсайта, web-приложение — это полноценная программа, доступ к которой пользователь получает через интернет, то есть она не требует установки на устройство. Web-приложение интерактивно и позволяет пользователям взаимодействовать с разными элементами: например, оставить заявку на покупку товара, оформить покупку авиабилета или прокомментировать пост друга.

Web-приложение можно классифицировать по-разному: в зависимости от их функционала и назначения. Существуют три основных шаблона построения web-приложений:

MPA (multi-page application) – многостраничное приложение, которое отправляет запрос на сервер и полностью обновляет страницу, когда с ней завершается действие.

SPA (single-page application) – одностраничное приложение, содержащее HTML-страницу, которая динамически обновляется в зависимости от действия пользователя – без полной перезагрузки.

PWA (progressive web application) – приложение, которое пользователь может установить и использовать в оффлайн-режиме.

Архитектура SPA-приложений устроена так, что при первоначальном запуске посетитель видит основной контент сайта в браузере, а новые данные загружаются на ходу по мере необходимости, например, при прокрутке или клике на иконку. По сути одностраничные приложения во многом похожи на desktop-приложения (приложения, выполняющиеся в операционной системе): при переходе на новую страницу обновляется только часть контента, позволяя не загружать одни и те же элементы сайта множество раз.

MPA-приложения по принципу работы полностью противоположны SPA. Это многостраничные приложения, работающие как привычные web-сайты. Они отправляют запрос на сервер и полностью обновляют страницу, когда с

ней совершается какое-либо действие (переход на другую страницу, внесение и изменение данных). Подобная архитектура приложения значительно влияет на скорость и производительность, поскольку большая часть данных подгружается повторно при каждом переходе.

PWA является чем-то средним между сайтом и приложением. Чтобы начать работу с PWA, необходимо сначала скачать и установить сайт как приложение. Последующий доступ к сайту производится через иконку на рабочем столе пользователя. При нажатии сайт быстро открывается без посредника в виде браузера.

Также существуют другие классификации web-приложений, основанных на их предназначении:

- Корпоративные порталы – позволяют автоматизировать многие бизнес-процессы с помощью одного продукта. Корпоративный портал позволяет работать с документами, отслеживать работу сотрудников, общаться с контрагентами, проводить PR-мероприятия, связывать подразделения компании и так далее.

- CRM (customer relationship management) – позволяют настроить воронку продаж, управлять взаимоотношениями с клиентами, содержать клиентскую базу и сократить документооборот.

- ERP (enterprise resource planning) – позволяет стандартизировать формы отчетности, контролировать бизнес-процессы, улучшить взаимодействие отделов и интегрировать контрагентов в рабочий процесс.

- Системы электронной коммерции – позволяет клиенту детально представить продукт, принимать заявки и продавать товары или услуги.

Web-приложения работают по принципу «клиент-сервер». Клиент-браузер связывается с веб-сервером посредством сети. Содержание web-приложения на устройстве пользователя формируется, когда он отправляет определенный запрос.

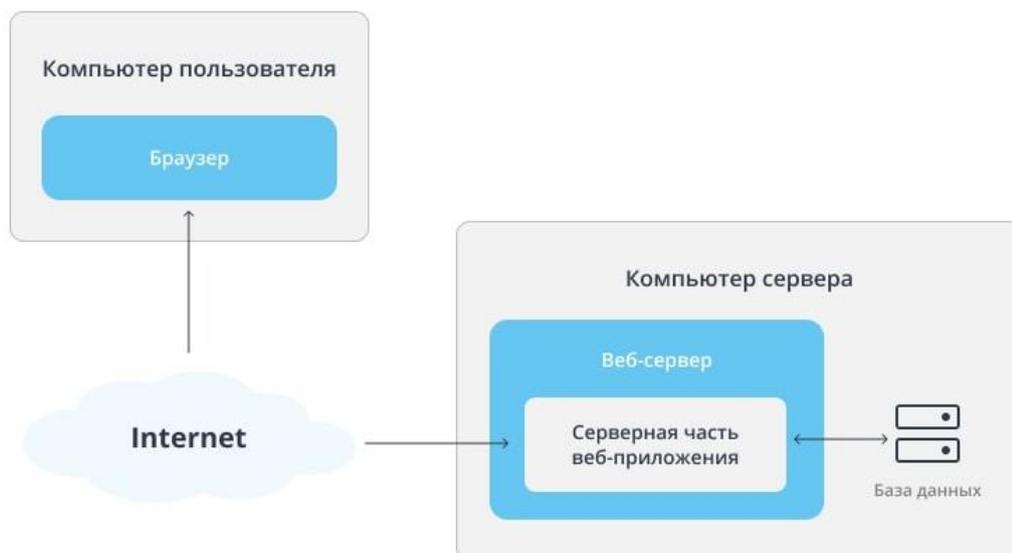


Рис. 1.1 Принцип работы web-приложений

В зависимости от типа web-приложения принципы их работы могут отличаться:

- Статические страницы – пользователь делает запрос в браузере, а web-сервер обрабатывает его и отправляет в ответ заранее созданную web-страницу. Это может быть, например, новостной материал или другие данные, которые не зависят от действий пользователя.

- Динамические страницы – не отправляются напрямую от web-сервера браузеру. Сначала они направляются на сервер приложений, где считывается код и подбираются данные для формирования страницы. Только после этого страницу отправляется на web-сервер, а затем клиенту-браузеру.

Web-приложения имеют следующие преимущества:

- Экономия – в ходе разработки не придется создавать отдельные приложения для разных операционных систем – они одинаково работают в различных современных браузерах: Opera, Safari, Google Chrome.

- Безопасность – web-система имеет единую точку входа, поэтому есть возможность централизованно настроить ее защиту. Кроме того, данные пользователей хранятся в облаке, поэтому при повреждении жесткого диска информация уцелеет.

- Доступ с различных устройств – пользователь может

взаимодействовать с одним web-приложением через смартфон, планшет или компьютер.

- Отсутствие клиентского ПО – пользователям не нужно ничего скачивать и обновлять. Можно произвести замену интерфейса, а обновление произойдет при очередной загрузке страницы.

- Масштабируемость – не придется наращивать мощность клиентских мест, если нагрузка на систему увеличится. Обычно web-приложения могут обрабатывать большое количество данных силами аппаратных ресурсов.

В основе функционирования web-приложений лежит такое понятие как web-сервер. Web-сервер – это программа, которая принимает входящие HTTP-запросы, обрабатывает их, генерирует HTTP-ответ и отправляет клиенту-браузеру. Общий алгоритм работы web-сервера можно представить в виде следующего рисунка (зеленым цветом помечены действия, обрабатываемые web-сервером).

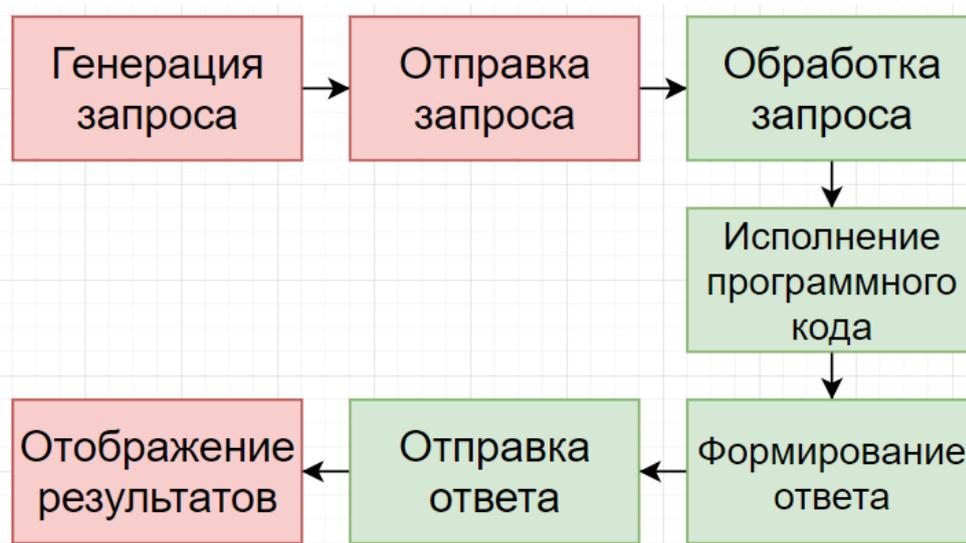


Рис. 1.2. Общий алгоритм работы web-сервера

После того, как пользователь обратился к определенному ресурсу по протоколу HTTP, браузер клиента формирует HTTP-запрос к web-серверу. Обычно указывается символическое имя сервера (например, <http://www.mysite.com>) – в этом случае браузер предварительно преобразует это имя в IP-адрес при помощи сервисов DNS. После этого по протоколу HTTP на web-сервер отправляется сформированное HTTP-сообщение. В этом сообщении

браузер указывает какой ресурс необходимо загрузить и всю дополнительную информацию. Задача web-сервера – прослушивание определенного TCP-порта и прием всех входящих HTTP-сообщений. Если входящие данные не соответствуют формату сообщения HTTP, то такой запрос игнорируется, а клиенту возвращается сообщение об ошибке.

При поступлении HTTP-запроса web-сервер считывает содержимое запрашиваемого файла с жесткого диска, упаковывает его содержимое в HTTP-ответ и отправляет браузеру клиента. В случае, если требуемый файл не найден, то web-сервер сгенерирует ошибку с указанием статусного кода 404 и отправит это сообщение браузеру клиента. Такой вариант работы web-сервера называют статическими сайтами. В этом случае на стороне сервера не запускается никакой программный код, кроме программного кода самого web-сервера. Но также бывают приложения, HTML-документы и другие ресурсы которого не хранятся на сервере в виде статичных файлов. Вместо них на сервере хранится программный код, который способен сгенерировать эти данные в момент обработки запроса. Разумеется, некоторые ресурсы могут храниться как статическое содержимое (например, картинки, файлы и т.д.), но основные HTML-страницы генерируются в процессе обработки программного кода, к которому обращается web-сервер. Данный алгоритм работы можно представить в виде следующей блок-схемы.

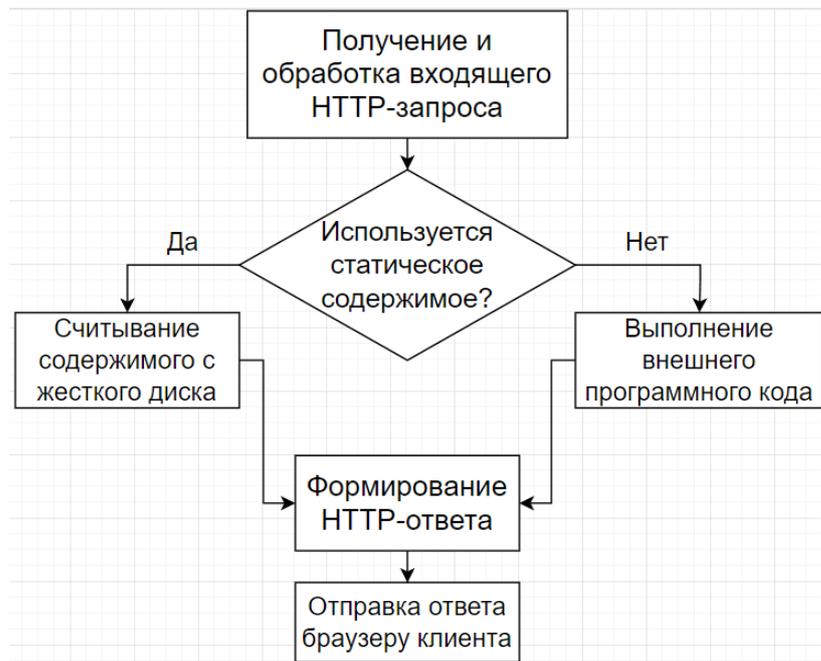


Рис. 1.3. Блок-схема алгоритма работы web-сервера

Одной из наиболее важных задач, которые решаются при построении web-сервера является задача обеспечения масштабируемости и защищенности от внешних атак. Поскольку web-сервер работает в открытой среде – глобальной сети Internet – то зачастую доступ к нему может осуществляться откуда угодно. Это делает web-сервер подверженным большим нагрузкам и потенциальным атакам. Наиболее распространенными атаками на web-сервер является обращение к web-серверу с большим количеством запросов и их высокой частотой. В этом случае web-сервер не сможет быстро обрабатывать все запросы, а это сказывается на производительности web-сервера для настоящих пользователей. Особенно остро подобным атакам подвержены web-сервера, на которых исполняется какой-то внешний программный код за исключением программного кода самого web-сервера. Обычно для борьбы с подобными атаками блокируются все запросы, которые приходят с определенного IP-адреса. Также в подобных случаях следует позаботиться об использовании кэширования – в этом случае при обработке каждого запроса нагрузка на центральный процессор будет меньше.

Исторически сложилось так, что существует два главных типа интерфейса взаимодействия внешнего приложения и web-сервера – CGI и ISAPI.

CGI (Common Gateway Interface) – наиболее ранний способ взаимодействия web-сервера и web-приложения. Основная идея, лежащая в основе CGI, заключается в том, что при поступлении очередного HTTP-запроса, web-сервер инициирует создание нового процесса и передает ему все необходимые данные HTTP-запроса. После того, как этот процесс завершится, он передает результат работы обратно web-серверу.

ISAPI (Internet Server API) – альтернативный способ взаимодействия web-сервера и web-приложения. При взаимодействии в рамках интерфейса ISAPI, при поступлении очередного запроса, web-сервер инициирует создание нового потока в рамках основного процесса, в котором работает web-сервер. Так как с точки зрения операционной системы создание потока – это менее затратная операция, чем создание процесса, то такие приложения на практике оказываются более масштабируемыми.

Таким образом, для разработки web-приложения, позволяющего создавать образовательные чат-боты отлично подойдет архитектура SPA-приложения и интерфейс взаимодействия внешнего приложения с web-сервером – ISAPI.

1.2. Инструментальные средства разработки web-приложений для создания чат-ботов

После определения способов создания web-приложения для создания чат-ботов, необходимо определиться с работой web-приложений и их взаимодействие со сторонними сервисами.

Как правило, работа чат-бота состоит из следующих основных функциональных возможностей: (1) посредством HTTP-запросов бот и клиент должны передавать друг другу сообщения, (2) для создания связи и получения ответов обеими сторонами необходим webhook (вебхук) [10], представленный URL. Он в свою очередь состоит из двух элементов: фронтэнда и бэкэнда.

Фронтэнд (front-end) является открытой для глаз посетителя частью сервиса. То есть это то, что пользователь видимо в браузере и может совершать

какие-либо действия.

Бэкэнд (back-end) скрыт от глаз клиента. Это совокупность средств, которая помогает сайту функционировать (получение данных из баз данных и возврат ответа во фронтэнд).

В качестве основного инструмента для создания чат-бота используются модели машинного обучения, либо онлайн-конструкторы. При использовании машинного обучения, чат-бот поймет, о чем пишет пользователь. Современные подходы – например, BERT (Bidirectional Encoder Representations from Transformers – двунаправленное представление кодировщика от трансформаторов) – находятся в открытом доступе. Но обычно требуется много доработок, под конкретный домен, язык пользователей и окружение, в котором происходит общение. Есть также готовые платформы, которые сразу пригодны для внедрения, но не всегда их гибкости достаточно [5].

AI (Artificial Intelligence) – чат-боты также делятся на несколько типов. Поисковые чат-боты применяют эвристические методы для выбора ответа из библиотеки predetermined реплик. Такие боты используют текст сообщения и контекст диалога для выбора ответа из predetermined списка. Контекст включает в себя текущее положение в дереве диалога, все предыдущие сообщения и сохраненные ранее переменные (например, имя пользователя). Эвристика для выбора ответа может быть спроектирована по-разному: от условной логики до машинных классификаторов; генеративные – могут самостоятельно создавать ответы и не всегда отвечают одним из predetermined вариантов. Это делает их более интеллектуальными, так как такие боты изучают каждое слово в запросе и генерируют ответ.

Чат-боты на основе меню относятся к наиболее распространенным видам чат-ботов. Они работают на основе иерархической древовидной структуры, но при этом пользователь предоставляет больше входных данных для определения подходящего ответа. Виртуальные чат-боты представляют собой кнопки, включающие меню и подменю на основе родительского меню, выбранного

пользователем. Они хорошо работают для ответов на простые вопросы, но не могут быть предпочтительными для аргументации или логических вопросов.

Также существуют чат-боты на основе ключевых слов: они читают вопрос, заданный пользователем, и фиксируют основные ключевые слова, размещенные в нем. Они умнее, чем боты, основанные на меню, и пытаются ответить на вопросы более правильно. Например, если пользователь задал 23 вопрос «Как создать новую учетную запись?», то чат-боты такого типа фиксируют ключевые слова «Как», «Создать» и «Новая учетная запись», сверяются с источником данных, получают информацию по теме «создание учетной записи» и передают ее пользователю. Тем не менее, такие AI-боты имеют свои ограничения и часто ошибаются там, где ключевые слова совпадают в разных вопросах. Следовательно, их не удастся эффективно использовать для всех целей.

Контекстные чат-боты считаются самыми умными и логичными AI-чат-ботами. Эти боты представляют собой комбинацию машинного обучения и искусственного интеллекта. Они многому учатся и фиксируют поведение пользователя: то, какие вопросы он задает и как именно. Тем не менее, обучение занимает некоторое время, в самом начале общения с определенным пользователем такие чат-боты могут работать неидеально, но по мере обучения их взаимодействие с пользователем становится гораздо лучше, чем в случае с ботами на основе меню или ключевых слов. Например, если вы обычно с помощью контекстного чат-бота заказываете некий товар с определенными свойствами, то бот запомнит ваш адрес и предпочтения, а в следующий раз, когда вы захотите заказать что-нибудь подобное, он предложит вам товар с похожими свойствами и сразу предложит отправить его по указанному ранее адресу [6].

Использование онлайн-конструкторов актуально, например, когда вам необходим чат-бот для какого-то мессенджера без использования сложных алгоритмов, чтобы пользователь мог получить актуальную информацию о

вашем товаре или услуге на основе команд или ключевых слов. Данный способ поможет избежать чрезмерных трат на разработку алгоритмов машинного обучения и времени на разработку чат-бота. Вам всего-навсего понадобится некоторое время для того, чтобы понять, как работает тот или иной онлайн-конструктор. Для создания такого онлайн-конструктора понадобится web-сайт, где будет размещено приложение, а также необходимо провести интеграцию с одним (или несколькими) мессенджерами.

Разработка архитектуры web-приложения является одним из ключевых моментов его создания, который в большой степени определяет эффективность его функционирования в будущем. С технической точки зрения архитектура приложения – это ни что иное, как все сторонние и внутренние приложения, с которыми будет происходить взаимодействие пользователя.

Для реализации web-приложения могут быть использованы следующие архитектуры [8]:

1. **Клиент-сервер** – данная архитектура представляет собой совокупность взаимодействующих компонент двух типов – клиентов и серверов. Клиенты обращаются к серверам с запросами, сервера их обрабатывают и возвращают результат. В этом случае в качестве сервера выступает СУБД, обеспечивающая выполнение запросов клиента, который в свою очередь реализует интерфейс пользователя.

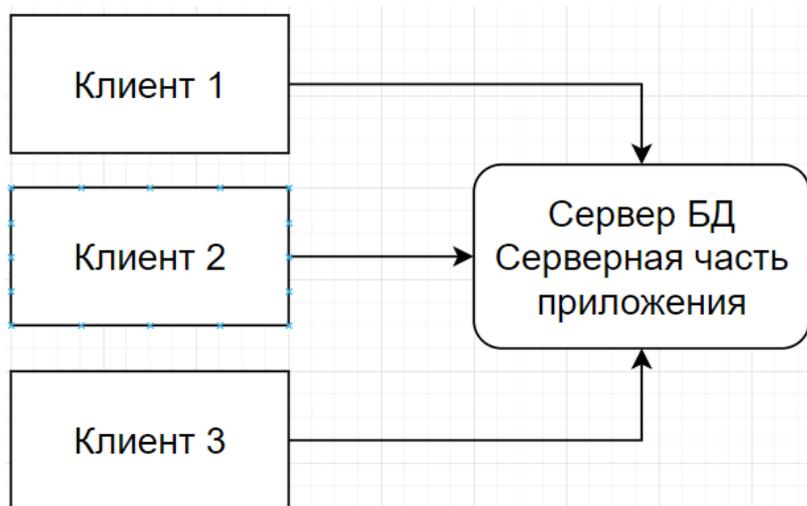


Рис. 1.4. Архитектура «Клиент-сервер»

2. Модель сервиса – представляет собой реализацию ситуации, при которой одну услугу реализует не один, а несколько серверов, представляемых клиенту как единое целое. Этот вариант отлично подойдет для критичных сервисов, для которых недопустимо приостановление обслуживания.

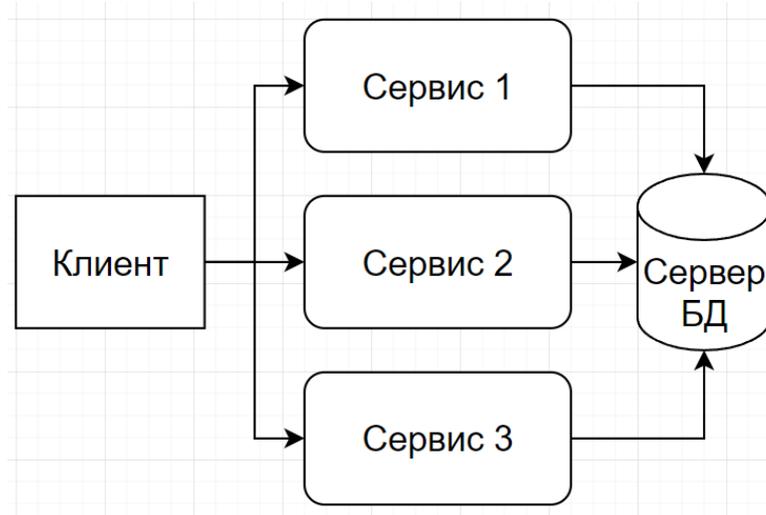


Рис. 1.5. Сервис-ориентированная архитектура

3. Тонкий клиент – суть технологии состоит в том, что клиент выполняет очень ограниченную по функционалу задачу (очень часто – только прием ввода с клавиатуры и других устройств и обработка команд рисования). Схема работы подобных систем: клиентская программа передает весь ввод пользователя (нажатия клавиш, движение мыши и т.д.) по сети серверу. Сервер, разбирает и обрабатывает этот ввод и передает клиенту готовые экраны, которые тот просто отображает.

4. Трехуровневая архитектура – почти то же самое, что и архитектура клиент-сервер, но только между клиентом и сервером появляется сервер бизнес-логики, который отвечает за выполнение запросов к СУБД и дальнейшей передачи ответа клиенту.

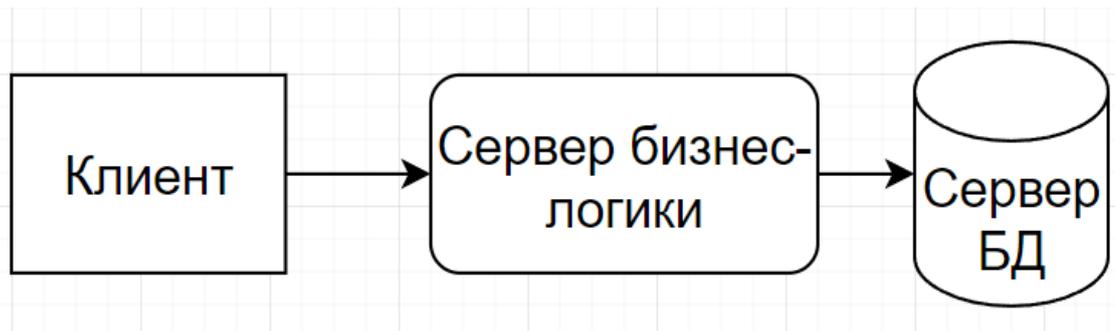


Рис. 1.6. Трехуровневая архитектура

Для создания web-приложения нужны разнообразные инструменты, которые помогут создать структуру, красиво оформить продукт и сделать его интерактивным. Основными технологиями для разработки web-приложений являются:

- HTML – стандартизированный язык разметки документов для просмотра веб-страниц в браузере. Веб-браузеры получают HTML документ от сервера по протоколам HTTP/HTTPS или открывают с локального диска, далее интерпретируют код в интерфейс, который будет отображаться на экране монитора [9].

- CSS – язык таблицы стилей для описания внешнего вида HTML-документа, который позволяет прикреплять стиль к структурированным документам.

- JavaScript – язык сценариев, который создавался для «оживления» web-страниц. Программы на этом языке называются скриптами и могут встраиваться в HTML и выполняться автоматически при загрузке web-страницы. Скрипты распространяются и выполняются как простой текст. Им не нужна специальная подготовка или компиляция для запуска.

- PHP – скриптовый язык программирования, применяемый для разработки web-приложений. Поддерживается большинством хостинг-провайдеров и является одним из лидеров среди языков, применяющихся для создания динамических web-сайтов.

- TypeScript – язык программирования, позиционируемый как средство

разработки web-приложений, расширяющее возможности JavaScript. Главной особенностью является строгая типизация переменных и объектов в JavaScript.

- Java – строго типизированный объектно-ориентированный язык программирования общего назначения.

- SQL – декларативный язык программирования, применяемый для создания и управления данными в реляционной базе данных, управляемой соответствующей СУБД.

Разработка web-приложений достаточно долгий и трудоемкий процесс. Основными этапами web-разработки являются:

- Постановка целей и задач приложения. Клиент определяет, зачем и какой продукт ему нужен. Это включает не только основные функции, но и глобальные цели – привлечь, обучить, продать и т.д.

- Проработка технического задания. Команда разработки плотно взаимодействует с заказчиком и определяет точные требования к финальному продукту.

- Прототипирование. Команда разработки создает прототип web-приложения и показывает клиенту примеры будущих web-страниц.

- Создание макета дизайна web-приложения. Дизайнеры разрабатывают внешний вид продукта и согласовывают результат с заказчиком.

- Разработка и верстка. Команда разработки полностью создает web-страницы, используя дизайн-макеты и техническое задание. Процесс делится на frontend и backend.

- Наполнение контентом. Команда разработки делает финальную работу над продуктом: добавляет нужный текст, картинки и видео на свои места.

- Тестирование. Тестировщики проверяют, что приложение работает корректно и все объекты отображаются нужным образом.

Для создания клиентской части лучше всего подойдет язык

программирования JavaScript, так как он универсален и без проблем выполняется в любом браузере.

Но в современной разработке web-приложений не обойтись без актуальной библиотеки фреймворка, в которых есть предварительно написанный код для использования в стандартных функциях и задачах программирования. Это основа для создания веб-сайтов или веб-приложений. Для этого могут быть использованы следующие фреймворки:

1. VueJS – это библиотека, позволяющая внедрять интерактивное поведение и дополнительные возможности в любой контекст, в котором выполняется JavaScript [11]. Vue можно использовать как на отдельных страницах, решая простые задачи, так и в качестве фундамента для полноценных промышленных приложений.

2. Angular – это фреймворк от компании Google для создания клиентских приложений. Прежде всего он нацелен на разработку SPA-решений (Single Page Application), то есть одностраничных приложений.

3. ReactJS – фреймворк от компании Facebook, которая используется для создания пользовательского интерфейса. Отличительная черта данной библиотеки – это создание отдельных компонентов приложения, которые в дальнейшем можно использовать в других проектах.

Для написания бэкэнд-части могут быть использованы такие технологии и языки программирования:

1. NodeJS – представляет среду выполнения кода на JavaScript, которая построена на основе движка JavaScript Chrome V8, который позволяет транслировать вызовы на языке JavaScript в машинный код. Прежде всего предназначен для создания серверных приложений.

2. ASP.NET Core WebAPI – представляет собой web-службу, которая может взаимодействовать с различными приложениями. При этом приложение может быть web-приложением ASP.NET, либо может быть мобильным или обычным десктопным приложением.

3. **Java Spring Framework** – универсальный фреймворк с открытым исходным кодом для Java-платформы. Используется в основном для больших приложений в крупных компаниях [12].

Часто при создании web-приложений используются СУБД (системы управления базами данных). Они разделяются на два класса разнородных СУБД: NoSQL, SQL.

SQL – реляционные системы управления базами данных. Основаны на реляционной модели, которая предполагает математический способ структуризации, хранения и использования данных. Отношения дают возможность группировки данных как связанных наборов, представленных в виде таблиц, содержащих упорядоченную информацию (например, имя и адрес человека) и соотносящих значения и атрибуты (его номер паспорта).

NoSQL – способ структуризации данных заключается в избавлении от ограничений при хранении и использовании информации. Базы данных NoSQL, используя неструктурированный подход, предлагают много эффективных способов обработки данных в отдельных случаях (например, при работе с хранилищем текстовых документов) [13].

Для того чтобы пользователь смог просматривать сайт в интернете, необходимо приобрести хостинг и доменное имя. После чего нужно определиться с web-сервером, который будет получать запросы от пользователя и обрабатывать их. Например, он может отдавать статический контент, такой как html-страница, медиа файлы, документы, архивы, картинки [14]. Наиболее популярными web-серверами являются Nginx, Apache, Tomcat, IIS.

При разработке сайта на локальном компьютере разработчика необходимо позаботиться о том, чтобы разрабатываемое приложение одинаково работало на всех хостингах независимо от установленной на них операционной системы. С этой задачей можно справиться при использовании технологии контейнеризации приложений, а помочь с этим может Docker.

Docker – это программное обеспечение для автоматизации развёртывания

и управления приложениями в средах с поддержкой контейнеризации, контейнеризатор приложений. Позволяет «упаковать» приложение со всем его окружением и зависимостями в контейнер, который может быть развернут на любой Linux-системе с поддержкой cgroups в ядре, а также предоставляет набор команд для управления этими контейнерами [15].

Контейнеры Docker могут разворачиваться и работать в любой среде. Контейнеры образов Docker работают в исходном формате в Linux и Windows. Но образы Windows будут выполняться только на узлах Windows, тогда как образы Linux – на узлах Linux или Windows.

Таким образом для создания front-end части web-приложения «Education-bot-creator» подойдут такие инструменты, как ReactJS и язык программирования TypeScript, Docker и Nginx. Для серверной части лучше использовать платформу NodeJS и язык программирования JavaScript, а для хранения данных – NoSQL-базу данных MongoDB.

1.3. Формализованное описание технического задания

ТЕХНИЧЕСКОЕ ЗАДАНИЕ

на разработку приложения-конструктор чат-ботов «Education-bot-creator»

Составлен на основе ГОСТ 34.602-89 «Техническое задание на создание автоматизированной системы» [16].

1. Общие сведения.

1.1. Название организации-заказчика.

ФГБОУ ВО «УрГПУ».

1.2. Название продукта разработки (проектирования).

Приложение-конструктор чат-ботов «Education-bot-creator».

1.3. Назначение продукта.

Создание чат-ботов на платформе VK обмена сообщениями с помощью web-интерфейса

1.4. Плановые сроки начала и окончания работ.

В соответствии с планом выполнения ВКР (01.03.2021 – 01.02.2022).

2. Характеристика области применения продукта.

2.1. Процессы и структуры, в которых предполагается использование продукта разработки.

Приложение конструктор чат-ботов «Education-bot-creator» будет предназначен для свободного использования в сети Интернет.

2.2. Характеристика персонала (количество, квалификация, степень готовности)

С продуктом будет работать пользователь с базовыми знаниями о работе с персональными компьютерами и веб-браузерами.

3. Требования к продукту разработки.

3.1. Требования к продукту в целом.

Чат-бот должен быть реализован на внешнем сервере с возможностью функционирования в социальной сети VK. Интерфейс создания чат-ботов должен позволять создавать:

- Ключевые слова для функционирования чат-бота.
- Диалоги для общения с пользователями в формате опросов и тестов с одиночным выбором правильного ответа.
- Группы подписчиков чат-бота.
- Автоматические рассылки сообщений для групп подписчиков чат-бота.

3.2. Аппаратные требования.

Процессор: x86- или x64-разрядный двухъядерный процессор с тактовой частотой 1,9 ГГц.

ОЗУ объемом 1 Гб.

Дисплей VGA с разрешением 1024x768

Пропускная способность сети более 50 КБ/с

Поддерживаемые web-браузеры: Microsoft Edge, Mozilla Firefox, Google Chrome.

3.3. Указание системного программного обеспечения (операционные системы, браузеры, программные платформы и т.п.).

Для данного приложения необходим любой веб-браузер. Приложение должно быть разработано с помощью любого JavaScript-фреймворка для Front-end и Back-end частей. СУБД – MongoDB.

3.4. Указание программного обеспечения, используемого для реализации.
ПО для разработки

Для разработки системы необходимо следующее ПО:

- Редактор исходного кода – Visual Studio Code с дополнениями Prettier и ESLint;
- Веб-браузер с поддержкой браузерных инструментов разработчика.

3.5. Форматы входных и выходных данных

Входные данные должны формироваться пользователем.

3.6. Источники данных и порядок их ввода в систему (программу), порядок вывода, хранения.

Данные должны формироваться пользователем с помощью ввода с клавиатуры, через веб-интерфейс.

3.7. Меры защиты информации.

Не предусмотрено.

4. Требования к пользовательскому интерфейсу.

4.1. Общая характеристика пользовательского интерфейса.

Дизайн должен быть выполнен в мягких тонах с использованием языка разметки HTML 5 и языка стилей CSS 3.

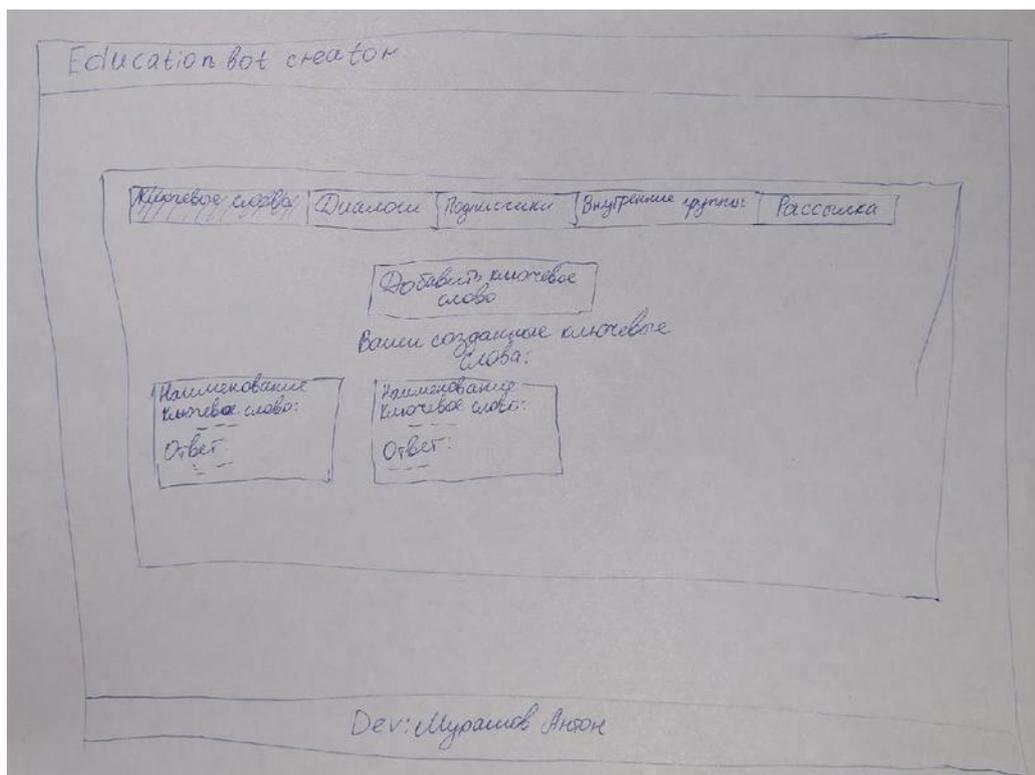


Рис. 1.7. Макет страницы «Личный кабинет»

Необходимо создать структуру сайта, состоящую из следующих элементов:

1. «Шапка» (хедер). В данном блоке необходимо расположить логотип и наименование сайта.
2. Блок отображения контента. Данный блок должен содержать перечень необходимых действий.
3. «Подвал» (футер). В данном блоке необходимо разместить краткую контактную информацию о разработчике и приложении.

Необходимо создать следующие страницы:

1. Главная страница. На данной странице необходимо расположить кнопку авторизации пользователя через API ВКонтакте.
2. Страница личного кабинета пользователя. На данной странице необходимо предоставить список сообществ, где авторизованный пользователь является администратором.
3. Страница редактирования действия в сообществе. На данной странице необходимо реализовать следующий функционал:

- a. Добавление, вывод и удаление ключевых слов, на которые сообщество должно отвечать пользователю в автоматическом режиме.
- b. Добавление и удаление диалогов с пользователем. Администратор сообщества должен иметь возможность строить диалог с помощью ввода сообщений и списка ответов. За каждый ответ должна быть возможность начислить баллы.
- c. Добавление и удаление внутренних групп, в которые администратор может добавить любого подписчика.
- d. Просмотр списка подписчиков и добавление нужных подписчиков во внутреннюю группу.
- e. Рассылки. С помощью данного функционала администратор сообщества должен иметь возможность отправить здесь и сейчас необходимое сообщение во внутренние группы.

Глава 2. Разработка web-приложения «Education-bot-creator»

2.1. Проектная часть web-приложения «Education-bot-creator»

Для того, чтобы начать разрабатывать web-приложение «Education-bot-creator», необходимо спроектировать его архитектуру, в которой необходимо отразить принцип работы пользователя с web-приложением, а также работу web-приложения с различными сторонними сервисами. Архитектура работы web-приложения «Education-bot-creator» приведена на рис. 2.1.

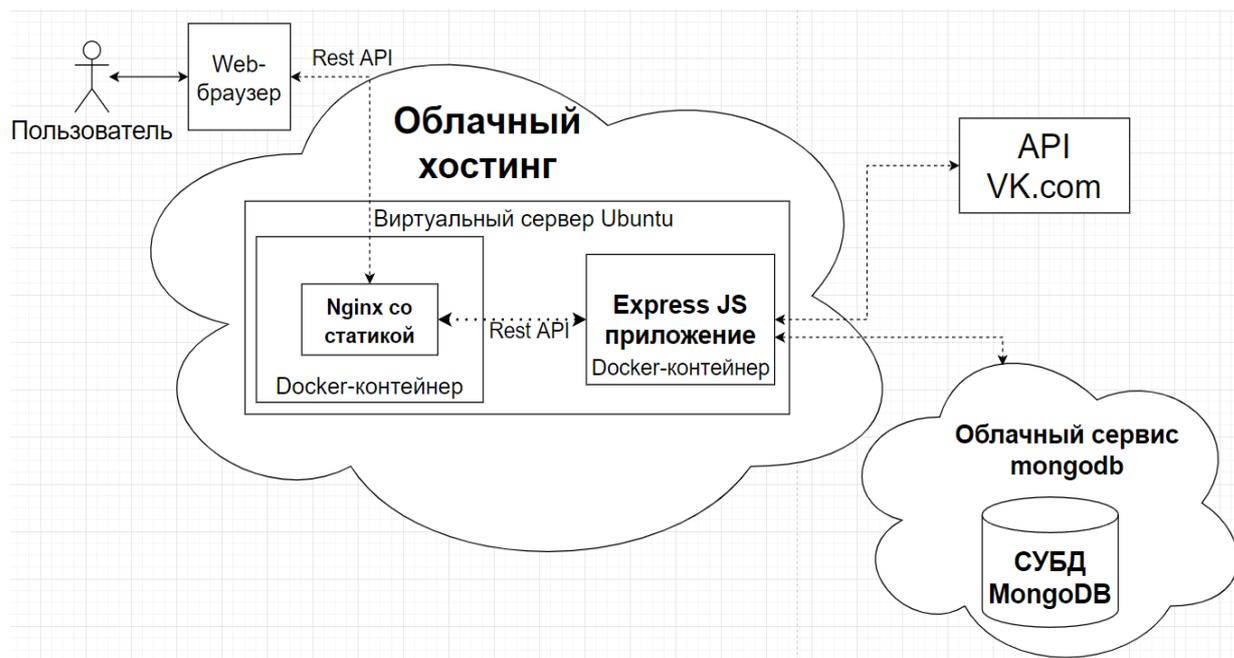


Рис. 2.1. Архитектура работы web-приложения «Education-bot-creator»

На данном рисунке пользователь, посредством Web-браузера обращается к виртуальному серверу Nginx, находящемуся на облачном хостинге, который возвращает статичные файлы приложения (JavaScript-файлы, HTML-страницы и таблицы стилей CSS) и отображает в браузере. Далее пользователь, взаимодействуя с front-end приложением в браузере, обращается посредством Rest API к back-end части. Back-end уже обращается, в зависимости от действий пользователя к API VK, либо к облачной базе данных Mongo DB.

Для создания front-end части необходимо использовать технологию одностраничных приложений, именуемой SPA (Single Page Application – приложение одной страницы). Суть технологии в том, что пользователю возвращается с веб-сервера единственная страница (чаще всего это index.html), которая содержит минимальную разметку, написанной на HTML с использованием div-контейнера, в который, с помощью API JavaScript подгружаются созданные разработчиком компоненты. Для создания компонентов на языке JavaScript отлично подходит фреймворк React JS. Данный фреймворк использует JSX – расширение языка JavaScript, которое позволяет писать HTML-разметку в JavaScript-файлах. Таким образом, файл одного компонента будет состоять из директив, написанных на JavaScript и HTML-разметки.

Для создания back-end части необходимо использовать программную платформу NodeJS и библиотеку ExpressJS. Данная библиотека содержит в себе заготовку почти-что готового начального каркаса back-end приложения, которое нужно всего лишь запустить через консоль операционной системы.

Для хранения информации используется облачная NoSQL-СУБД MongoDB. Данное решение имеет ряд преимуществ:

1. Масштабирование по мере необходимости.
2. Контроль расходов.
3. Доступ к данным из любого места.
4. Снижение затрат за счет консолидации ресурсов.

Rest API – это способ взаимодействия сайтов и веб-приложений с сервером. Его также называют RESTful [17]. Rest API применяется везде, где пользователю сайта или веб-приложения нужно предоставить данные с сервера. Например, при нажатии на кнопку сохранения Rest API может отправить какие-то данные, введенные пользователем на сервер. В настоящий момент Rest API – наиболее популярный архитектурный стиль взаимодействия клиентской части приложения с серверной.

Также необходимо проработать архитектуру взаимодействия приложения со сторонним API. В данном приложении сторонним API выступает мессенджер VK. Для работы с методами API необходимо в каждом методе передавать специальный ключ, который представляет собой строку из латинских букв и цифр и может соответствовать отдельному пользователю, сообществу или приложению. Для получения ключа доступа используется открытый протокол OAuth 2.0. При использовании данного протокола пользователь не передает логин и пароль приложению, поэтому его аккаунт не может быть скомпрометирован.

API VK поддерживает три способа получения ключа доступа по OAuth 2.0:

1. **Implicit flow** – самый короткий и простой вариант. Ключ возвращается на устройство пользователя, где был открыт диалог авторизации. Такой ключ может быть использован только для запросов непосредственно с устройства пользователя.

2. **Authorization code flow** – двухэтапный вариант с дополнительной аутентификацией сервера. Ключ доступа возвращается непосредственно на сервер и может быть использован, например, для автоматизированных запросов из JavaScript-приложения.

3. **Client credentials flow** – авторизация по секретному ключу приложения. Этот подход необходимо использовать только для доступа к специальным **secure**-методам.

Для получения ключа доступа необходимо использовать способ Authorization code flow, так как для запросов к API будет использоваться серверное приложение. Для этого необходимо перенаправить браузер пользователя по адресу <https://oauth.vk.com/authorize>, передав следующие параметры:

Таблица 2.1

Параметры, передаваемые в запросе <https://oauth.vk.com/authorize>

| Наименование параметра | Описание |
|------------------------|--------------------------|
| client_id | Идентификатор приложения |

| | |
|-------------------------------------|---|
| обязательный | |
| redirect_uri обязательный | Адрес, на который будет переадресован пользователь после прохождения авторизации |
| display | <p>Указывает тип отображения страницы авторизации. Поддерживаются следующие варианты:</p> <ul style="list-style-type: none"> • page — форма авторизации в отдельном окне; • popup — всплывающее окно; • mobile — авторизация для мобильных устройств (без использования Javascript) <p>Если пользователь авторизуется с мобильного устройства, будет использован тип mobile.</p> |
| scope | Битовая маска настроек доступа приложения, которые необходимо проверить при авторизации пользователя и запросить отсутствующие. |
| response_type | Тип ответа, который необходимо получить. Необходимо указать token |
| state | Произвольная строка, которая будет возвращена вместе с результатом авторизации |
| revoke | Параметр, указывающий, что необходимо не пропускать этап подтверждения прав, даже если пользователь уже авторизован |

Разработанная диаграмма последовательности (sequence diagram) отображает взаимодействие объектов при выполнении авторизации (рис. 2.2) с использованием протокола OAuth 2.0.

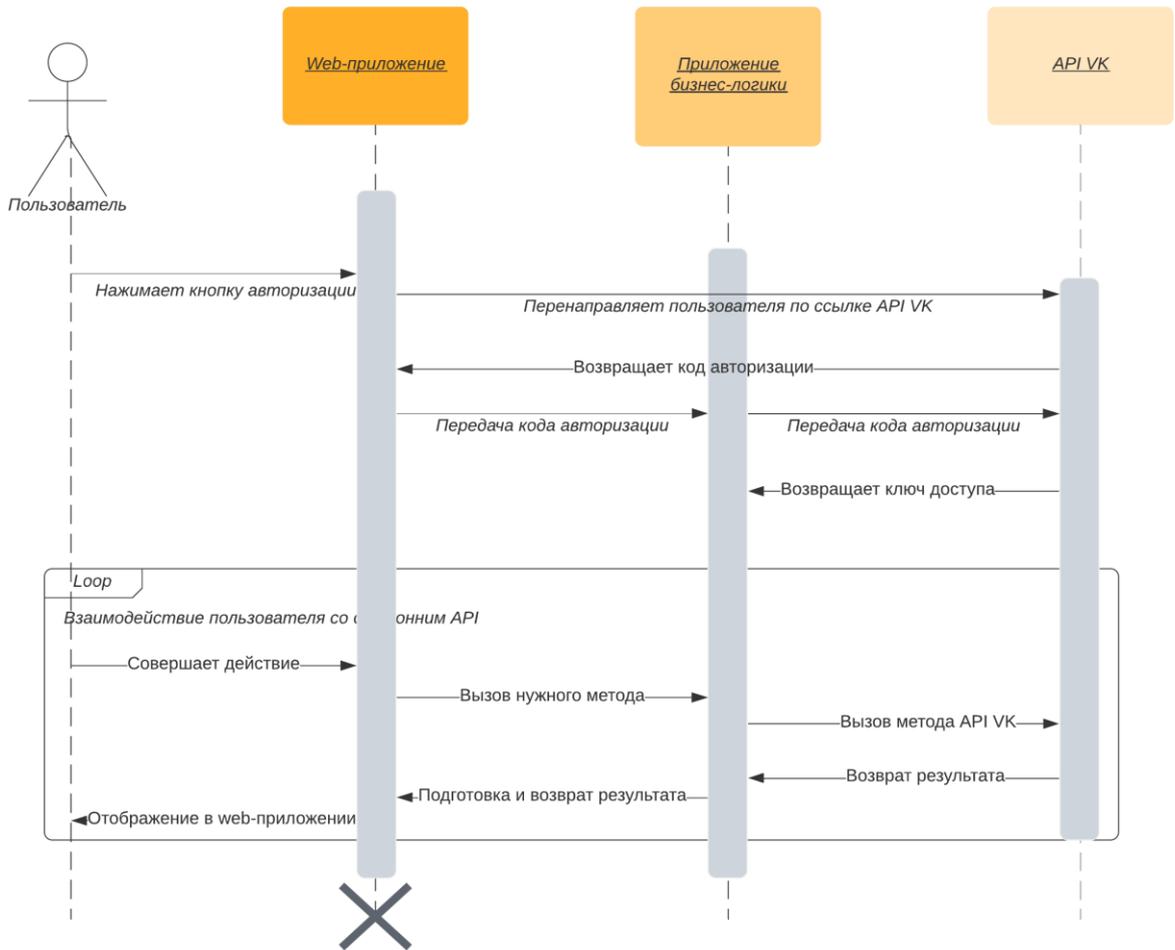


Рис. 2.2 Диаграмма последовательности

После успешного входа на сайт пользователю будет предложено авторизовать приложение, разрешив доступ к необходимым настройкам, запрошенным при помощи параметра **scope**. Для пользователя никакие права не требуются, поэтому этот параметр можно опустить.

После успешной авторизации приложения браузер пользователя будет перенаправлен по адресу **redirect_uri**, указанному при открытии диалога авторизации. При этом ключ доступа к API **access_token** и другие параметры будут переданы в URL-фрагменте ссылки: `http://REDIRECT_URI#access_token=533bacf01e11f55b536a565b57531ad114461ae8736d6506a3&expires_in=86400&user_id=8492&state=123456`

Вместе с ключом **access_token** также будет указано время его жизни **expires_in**, заданное в секундах. **expires_in** содержит 0, если токен

бессрочный (при использовании **scope = offline**). Если срок использования ключа истек, то необходимо повторно провести все описанные выше шаги, но в этом случае пользователю уже не придется дважды разрешать доступ. Запрашивать **access_token** также необходимо при смене пользователем логина или пароля, или удалением приложения в настройках доступа.

Кроме того, среди возвращаемых параметров будет указан **user_id** – идентификатор авторизовавшегося пользователя.

В случае возникновения ошибки авторизации в качестве GET-параметров в **redirect_uri** будет передана информация об этой ошибке.

После авторизации пользователя, необходимо получить ключ доступа сообщества для работы с API от имени группы, встречи или публичной страницы. Например, с его помощью можно отвечать подписчикам сообщества на сообщения, поступившие в его адрес. Для получения ключа доступа сообщества есть также несколько способов: **Implicit flow** и **Authorization code flow** (такие же, как и при получении ключа доступа пользователя). Тут также необходимо использовать способ Authorization code flow. Для этого необходимо выполнить запрос с серверного приложения на https://oauth.vk.com/access_token, передав следующие параметры:

Таблица 2.2

Параметры, передаваемые в запросе https://oauth.vk.com/access_token

| Наименование параметра | Описание параметра |
|------------------------|---|
| client_id | Идентификатор приложения |
| client_secret | Защищенный ключ приложения |
| redirect_uri | URL, который использовался при получении code на первом этапе авторизации. Должен быть аналогичен переданному при авторизации |
| code | Временный код, полученный после прохождения авторизации |

Пример запроса:

https://oauth.vk.com/access_token?client_id=1&client_secret=H2Pk8htyFD80

24mZaPHm&redirect_uri=http://mysite.ru&code=7a6fa4dff77a228eeda56603b8f53806c883f011c40b72630bb50df056f6479e52a%3C/b%3E

Пример ответа:

```
{
  "access_token_123456":
"533bacf01e11f55b536a565b57531ac114461ae8736d6506a3",
  "access_token_654321":
"a740d2bfe91caaa6eab794e1168da38cdaedc93c92f233638f",
  "groups": [
    {
      "group_id": 123456,
      "access_token":
"533bacf01e11f55b536a565b57531ac114461ae8736d6506a3"
    },
    {
      "group_id": 654321,
      "access_token":
"a740d2bfe91caaa6eab794e1168da38cdaedc93c92f233638f"
    }
  ],
  "expires_in": 0
}
```

После проделанных действий будут доступны методы API VK для выполнения запросов. Каждый метод выполняется на серверах VK. Для этого необходимо отправить GET или POST запрос. Пример вызова метода:

```
https.get(
  `https://api.vk.com/method/groups.getCallbackServers?group_id=${gro
up_id}&access_token=${token_group}&v=5.131`
```

В этом примере вызывается метод `getCallbackServers` из группы `groups` через GET-запрос. В него передаются параметры `group_id` и `access_token`. Если параметры верны, то сервер вернет необходимые данные, в ином случае вернется ошибка.

Таким образом, можно сделать вывод, что общение между данным серверным приложением и VK происходит посредством вызова методов API VK через GET и POST-запросы.

Для хранения данных используется NoSQL база данных MongoDB. Данная база данных не оперирует понятием «таблица». Вся информация хранится в коллекциях, которые представляют собой JSON-схему с необходимыми полями. Для приложения необходимо создать следующие коллекции:

Таблица 2.3

Коллекции MongoDB для хранения данных web-приложения
«Education-bot-creator»

| Наименование коллекции | Описание |
|------------------------|--|
| keywords | Хранение ключевых слов |
| dialogs | Список созданных диалогов |
| dialogsStarting | Коллекция начатых диалогов с пользователем |
| messagesHistory | Коллекция истории сообщений, отправленных или полученных от пользователя |
| answersHistory | Коллекция истории ответов на вопросы в диалоге |
| dialogsHistory | Коллекция истории диалогов, которые были начаты с пользователем |
| innerGroups | Хранение внутренних групп |
| usersIngroups | Коллекция со списком пользователей, находящихся во внутренних группах |
| currentUserToken | Коллекция для хранения ключей доступа пользователя |
| currentGroupToken | Коллекция для хранения ключей доступа сообщества |
| mailings | Коллекция для хранения созданных рассылок |

Для обработки входящих сообщений от пользователя выбранному сообществу в мессенджере VK была составлена следующая блок-схема, представленная на рис. 2.3.

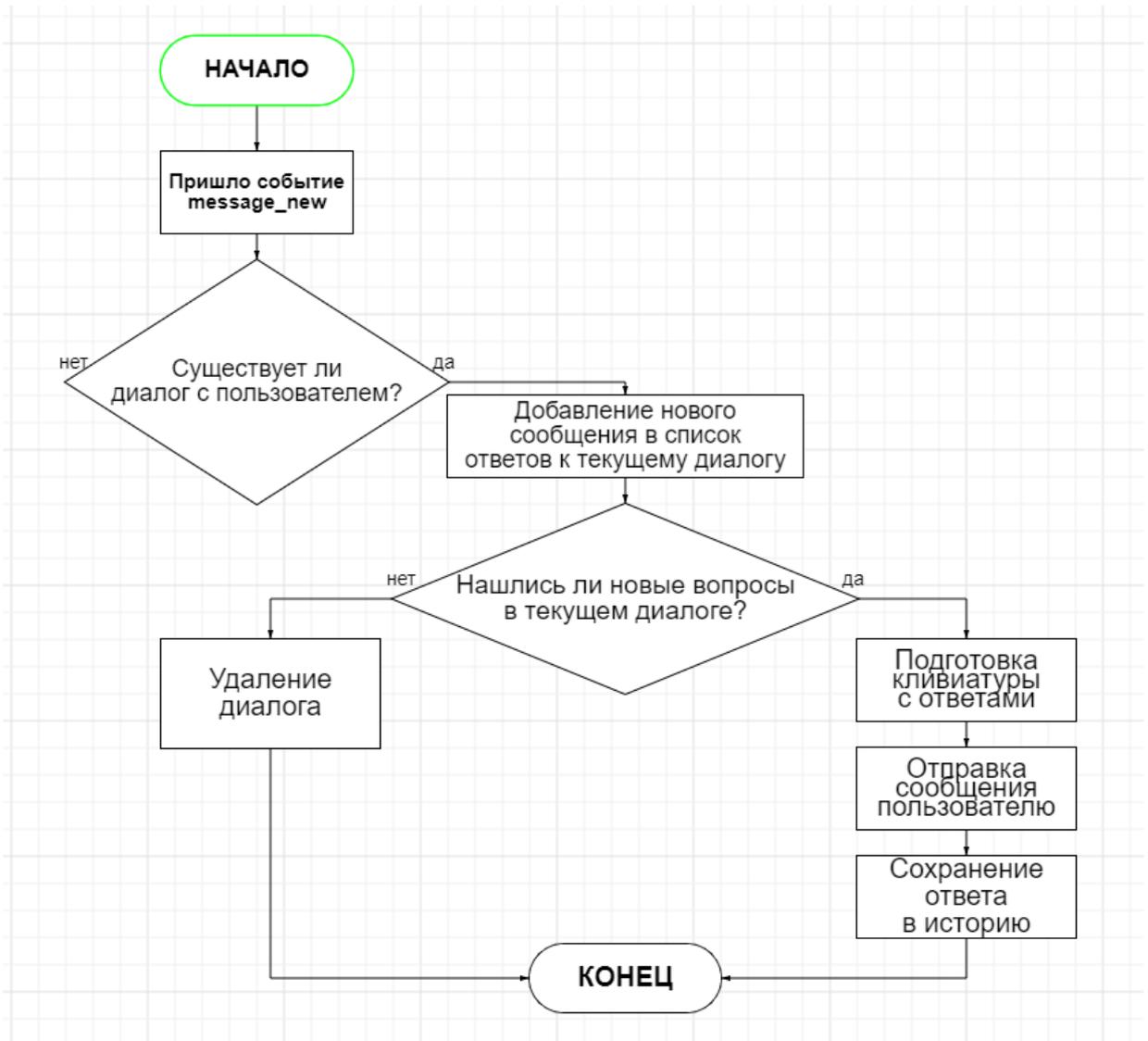


Рис. 2.3. Блок-схема обработки сообщений с существующим диалогом

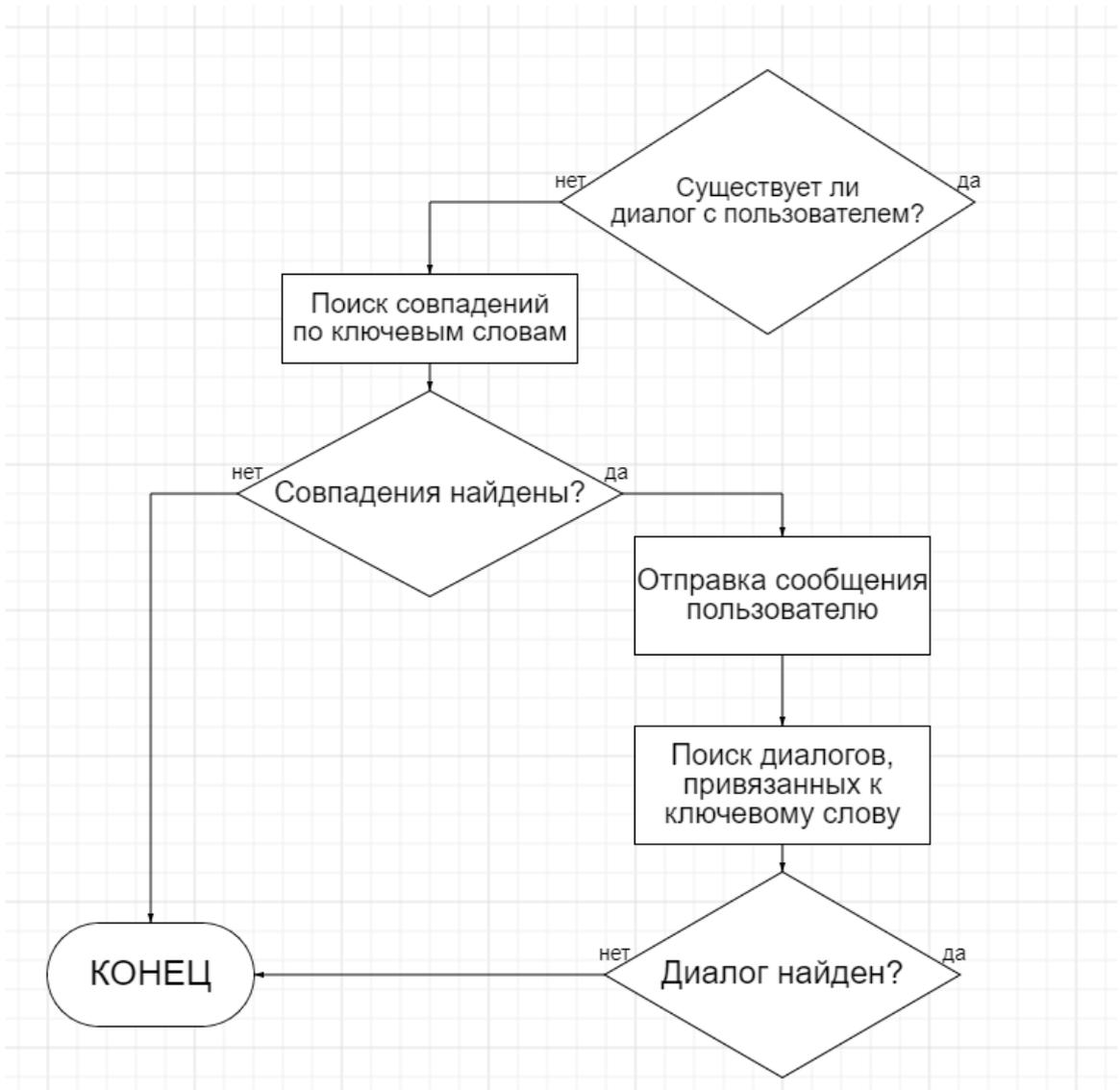


Рис. 2.4. Блок-схема обработки сообщений с существующим диалогом

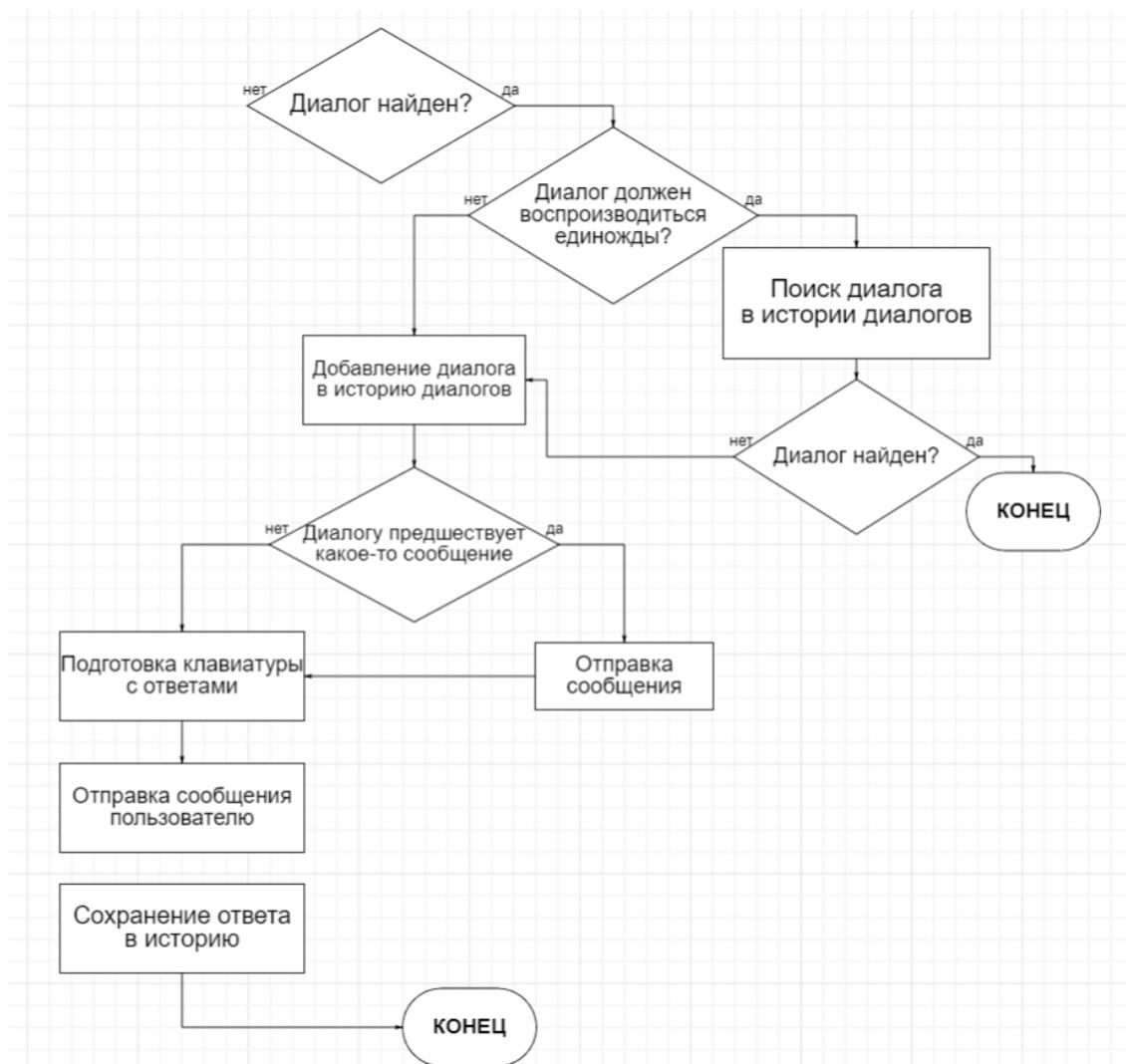


Рис. 2.5. Продолжение блок-схеме обработки сообщений с существующим диалогом

Таким образом, были спроектированы следующие элементы web-приложения «Education-bot-creator»:

- Для того, чтобы пользователь имел возможность работать с web-приложением, необходимо использовать web-сервер Nginx, который будет отправлять пользователю статические файлы web-приложения.
- Для backend-части приложения используется библиотека ExpressJS и платформа NodeJS, на которой backend-приложение будет запущено.
- Для взаимодействия web-приложения «Education-bot-creator» со сторонним мессенджером VK необходимо использовать вариант авторизации

Authorization code flow, который позволяет получить токен для выполнения запросов к серверам мессенджера VK.

- В качестве системы управления базами необходимо использовать NoSQL базу данных MongoDB.

2.2. Разработка основных элементов web-приложения «Education-bot-creator»

Для разработки UI-части приложения с использованием библиотеки React необходимо использовать утилиту create-react-app. Для этого в командной строке необходимо выполнить команду `npm create-react-app <folder_name> --typescript`, которая сперва установит утилиту create-react-app, а после чего начнет инициализацию проекта в папке `folder_name` с использованием шаблона TypeScript. После успешной инициализации, необходимо перейти в папку с установленным приложением с помощью команды `cd ./folder_name` и выполнить команду `npm start`. Приложение будет запущено в браузере и будет доступно по адресу `http://localhost:3000/`.

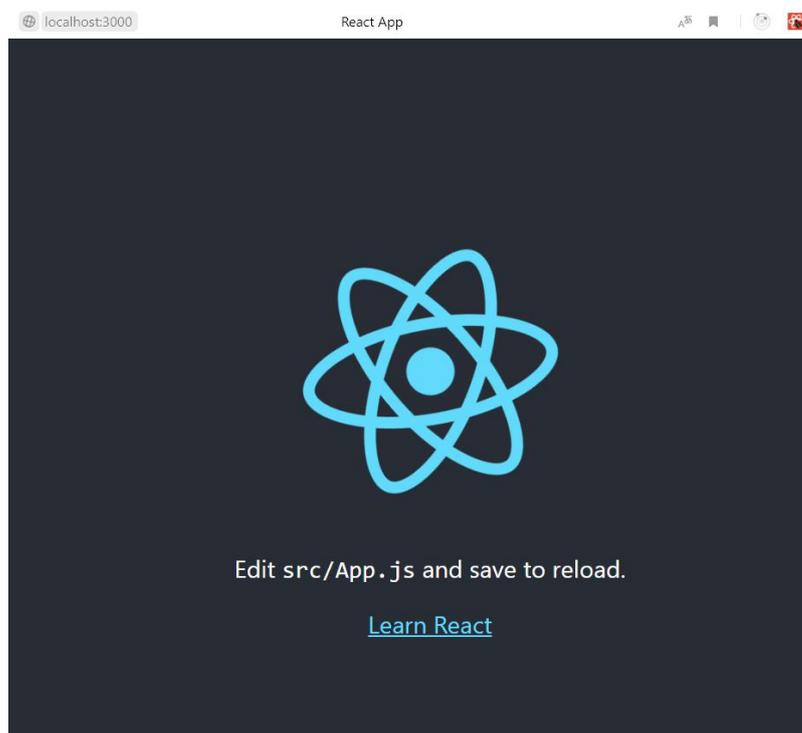


Рис 2.6. Запущенное стартовое приложение

После чего необходимо создать структуру папок приложения. Для этого в папке `src` были созданы следующие директории:

`Components` – директория компонентов, используемых в приложении

`Hooks` – Хуки

`Layouts` – шаблоны страниц

`Pages` – директория компонентов со страницами

`Store` – директория файлов для работы с хранилищем приложения

`Utils` – директория для вспомогательных методов, которые используются в приложении.

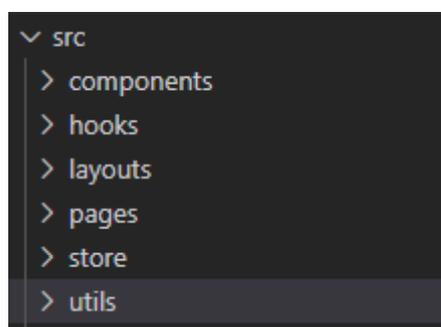


Рис. 2.7. Директории приложения

В качестве библиотеки компонентов используется пакет `@material-ui`, который содержит набор готовых стилизованных компонентов.

Для работы с состоянием приложения используется библиотека `redux`, вспомогательная библиотека `react-redux` и `middleware` (промежуточное ПО) `redux-saga`.

Для роутинга (работа с адресной строкой) используется библиотека `react-router-dom`.

Для выполнения AJAX-запросов используется библиотека `axios`.

После установки всех необходимых библиотек, необходимо внести изменения в файл `src/index.tsx`.

```

import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import { Router } from 'react-router-dom';
import { CssBaseline, ThemeProvider } from '@material-ui/core';

import App from 'pages/App';
import { theme } from 'utils/theme';
import history from 'utils/history';
import { store } from 'store';
import { RootLayout } from './layouts/RootLayout';

ReactDOM.render(
  <ThemeProvider theme={theme}>
    <Provider store={store}>
      <Router history={history}>
        <RootLayout>
          <App />
        </RootLayout>
      </Router>
      <CssBaseline />
    </Provider>
  </ThemeProvider>,
  document.getElementById('root')
);

```

Рис. 2.8. Стартовый файл index.tsx

В данном файле используются компоненты ThemeProvider для использования общей темы из библиотеки material-ui/core, компонент Provider для взаимодействия компонентов приложения с хранилищем redux. Компонент Router нужен для корректной работы роутинга. Далее идет компонент RootLayout, который содержит шаблон приложения.

```

const RootLayout: React.FC = ({ children }) => {
  const classes = useStyles();
  return (
    <div className={classes.wrapper}>
      <div className={classes.content}>
        <Header />
        <Container className={classes.root}>{children}</Container>
      </div>
      <div className={classes.footer}>
        <Footer />
      </div>
    </div>
  );
};

export default RootLayout;

```

Рис. 2.9. Компонент RootLayout

Реализация данного компонента позволяет на каждой странице использовать «шапку» и «футер» приложения без необходимости использовать компоненты Header и Footer в каждом другом компоненте приложения.

Компонент App – по сути входная точка в приложение, который

содержит роутинг. Его использование позволяет при изменении адресной строки отрисовывать тот или иной компонент.

```
const App = (): JSX.Element => {
  return (
    <Switch>
      <Route exact path="/" key="mainPage" component={MainPage} />
      <Route exact path="/office" key="officePage" component={OfficePage} />
      <Route
        exact
        path="/office/:id"
        key="editablePage"
        component={EditableGroupPage}
      />
      <Route
        exact
        path="/office/:id/new-message"
        key="newMessagePage"
        component={NewMessagePage}
      />
      <Route
        exact
        path="/office/:id/new-dialog"
        key="newDialogPage"
        component={NewDialogPage}
      />
      <Route
        exact
        path="/office/:id/new-inner-group"
        key="newInnerGroup"
        component={NewInnerGroupPage}
      />
      <Route
        exact
        path="/office/:id/new-mailing"
        key="newMailingPage"
        component={NewMailingPage}
      />
    </Switch>
  );
};

export default App;
```

Рис. 2.10. Компонент App

Компонент Switch необходим для того, чтобы объединить компоненты Route в одно целое.

Для создания стилей используется метод makeStyles из библиотеки @material-ui/core. Данный метод создает описанные стили с помощью JavaScript без необходимости создания отдельных файлов стилей css. Данная технология получила название css-in-js.

```
const useStyles = makeStyles<Theme>((theme) => ({
  wrapper: {
    display: 'flex',
    flexDirection: 'column',
    minHeight: '100vh',
  },
  content: {
    flex: '1 0 auto',
    backgroundColor: theme.palette.grey['50'],
  },
  footer: {
    flex: '0 0 auto',
  },
  root: {
    padding: 24,
    width: '100%',
  },
}));
```

Рис. 2.11. Метод для создания стилей

При первом заходе в приложение пользователю отрисовывается компонент MainPage, который содержит в себе, по сути, стартовую страницу.

```
const MainPage: React.FC = () => {
  const classes = useStyles();
  return (
    <Flexbox
      className={classes.root}
      justify="center"
      direction="column"
      align="center"
    >
      <div className={classes.authContainer}>
        <Auth />
      </div>
      <h3>
        Привет. Это web-приложение для создания чат-ботов для групп VK.com.
      </h3>
      <span className={classes.description}>
        Ты сможешь научить бота отвечать на ключевые слова, которые пользователи
        будут писать в группу.
      </span>
      <span className={classes.description}>
        Также ты сможешь создавать диалоги и делать рассылку пользователям
        внутренних групп.
      </span>
      <h4 className={classes.label}>
        На данный момент можно создавать только одного бота на группу
      </h4>
    </Flexbox>
  );
};

export default MainPage;
```

Рис. 2.12. Компонент стартовой страницы MainPage

Также в данном компоненте используется другой компонент Auth, который содержит код для авторизации пользователя.

```
const Auth: React.FC = () => {
  const classes = useStyles();
  const location = useLocation();
  const dispatch = useDispatch();

  React.useEffect(() => {
    const code = location.search.split('=')[1];
    if (code) {
      dispatch(AuthActions.vkAuth({ code }));
    }
  }, [dispatch, location.search]);

  return (
    <Flexbox
      className={classes.root}
      align="center"
      justify="center"
      direction="column"
    >
      <h4>Авторизоваться через:</h4>
      <a
        href={`https://oauth.vk.com/authorize?client_id=${process.env.VK_APP_ID}&redirect_uri=${process.env.REDIRECT_URI}`}
      >
        
      </a>
    </Flexbox>
  );
};

export default Auth;
```

Рис. 2.13. Компонент Auth

2.3. Пользовательский интерфейс web-приложения «Education-bot-creator»

В браузере первая страница web-приложения «Education-bot-creator» выглядят следующим образом:

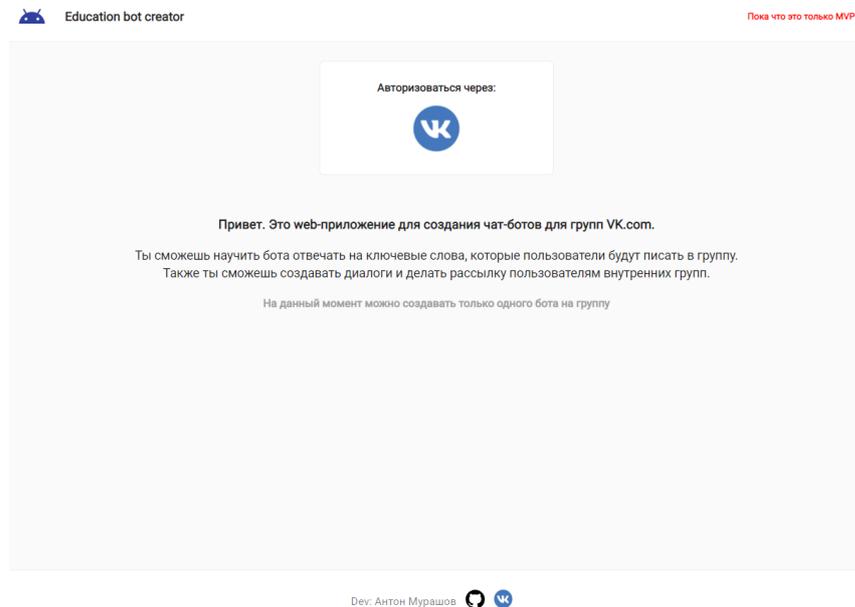


Рис. 2.14. Отображение стартовой страницы в браузере

После того, как пользователь нажмет на изображение VK, произойдет перенаправление пользователя по адресу:

```
https://oauth.vk.com/authorize?client_id=${process.env.VK_APP_ID}&redirect_uri=${process.env.REDIRECT_URI}&scope=email&groups&display=popup&response_type=code
```

После чего пользователь увидит страницу VK с предложением авторизоваться в используемом приложении.

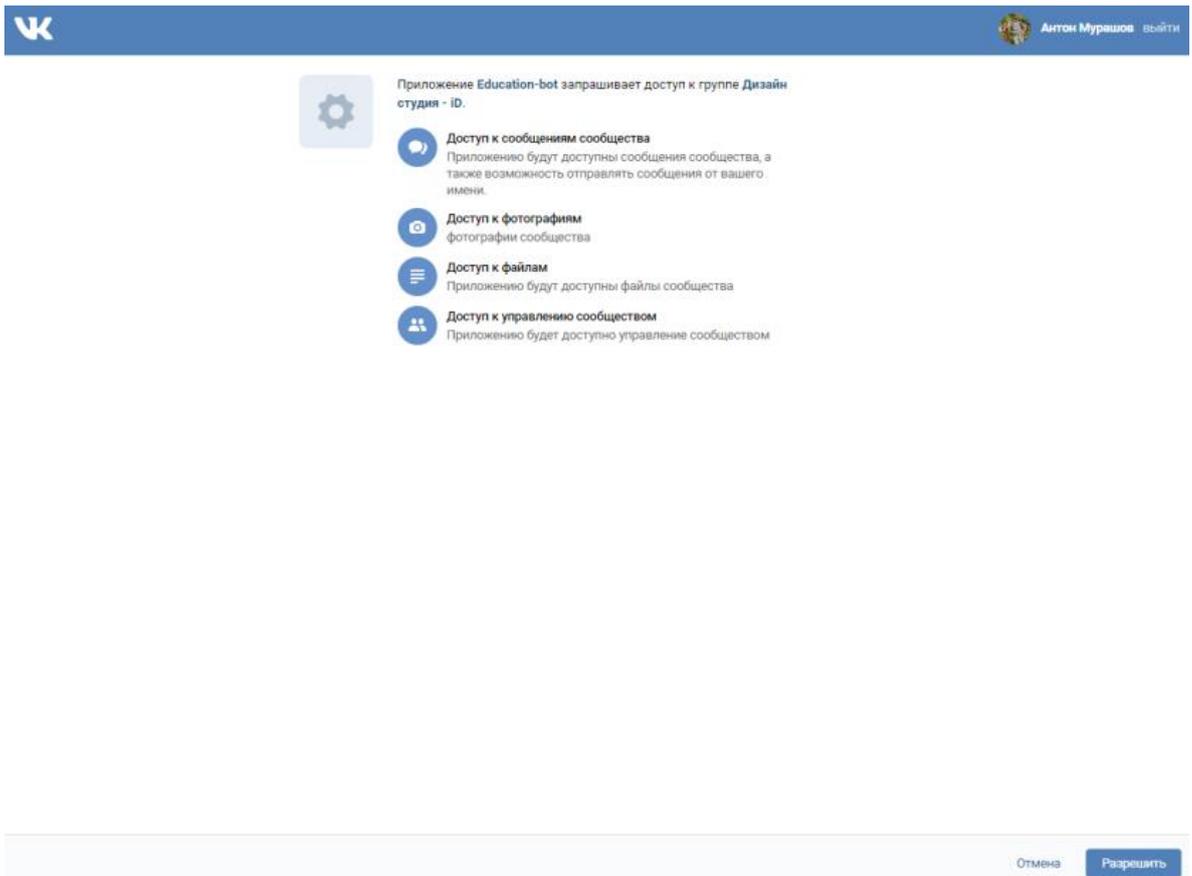


Рис. 2.15. Авторизация пользователя в приложении

После нажатия кнопки «Разрешить» произойдет перенаправление по адресу `redirect_uri`, который указан в переменной окружения `REDIRECT_URI`. Далее выполнится хук `useEffect` из компонента `Auth`, который инициирует выполнение метода `auth`, находящемуся на серверном приложении.

```
function* vkAuthWorker(action: { payload: { code: string } }) {
  const { code } = action.payload;
  try {
    const response: AxiosResponse<UserSettings> = yield call(() =>
      API.get('/auth', {
        params: {
          code,
        },
      })
    );
    if (response.status === 200) {
      document.cookie = `user_id=${response.data.user_id}`;
      history.push('/office');
      yield put(authActions.vkAuthSuccess({ payload: response.data }));
    }
  } catch {}
}
```

Рис. 2.16. Код метода выполнения запроса для авторизации

Данный запрос вызовет метод auth, код которого представлен на рис. 2.17.

```
app.get('/api/auth', async (req, res) => {
  const code = req.query.code;
  await https
    .get(
      `https://oauth.vk.com/access_token?client_id=${CLIENT_ID}&client_secret=${CLIENT_SECRET}&redirect_uri=${REDIRECT_URI}&code=${code}`
    )
    (responseVk) => {
      responseVk.on('data', async (d) => {
        const hasError = JSON.parse(d);
        if (hasError.error) {
          res.status(404).send(hasError);
          return;
        }
        const data = JSON.parse(d);
        const currentUserToken = {
          user_id: data.user_id.toString(),
          token: data.access_token,
          expires_in: data.expires_in * 1000,
          date_record: Date.now(),
          date_end: Date.now() + data.expires_in * 1000,
        };
        await app.locals.currentUserToken.deleteOne({
          user_id: data.user_id.toString(),
        });
        await app.locals.currentUserToken.insertOne(currentUserToken);
        res.status(200).send({ user_id: data.user_id, email: data.email });
      });
    }
  )
  .on('error', (e) => {});
});
```

Рис. 2.17. Код серверного метода auth

Данный метод выполняет запрос к API VK для получения access_token. После успешного выполнения пользователю возвращается объект с user_id, email. После чего в браузере происходит перенаправление пользователя по адресу /office и отрисовка следующей страницы – страницы выбора групп (компонент OfficePage), в которых пользователь является администратором. На рис. 2.18 представлен вывод этого компонента в браузере.

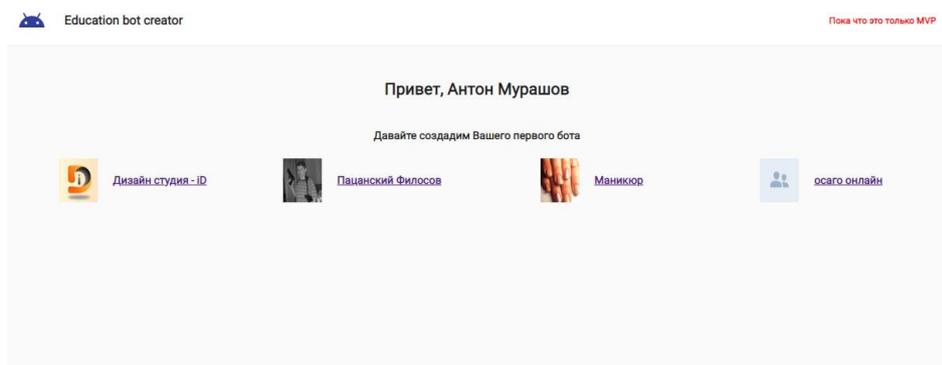


Рис. 2.18. Компонент OfficePage в браузере

После того, как пользователь выберет любую группу, браузер пользователя будет перенаправлен по адресу https://oauth.vk.com/authorize, в котором пользователю необходимо подтвердить свое намерение использовать данное приложение в своей группе или сообществе.

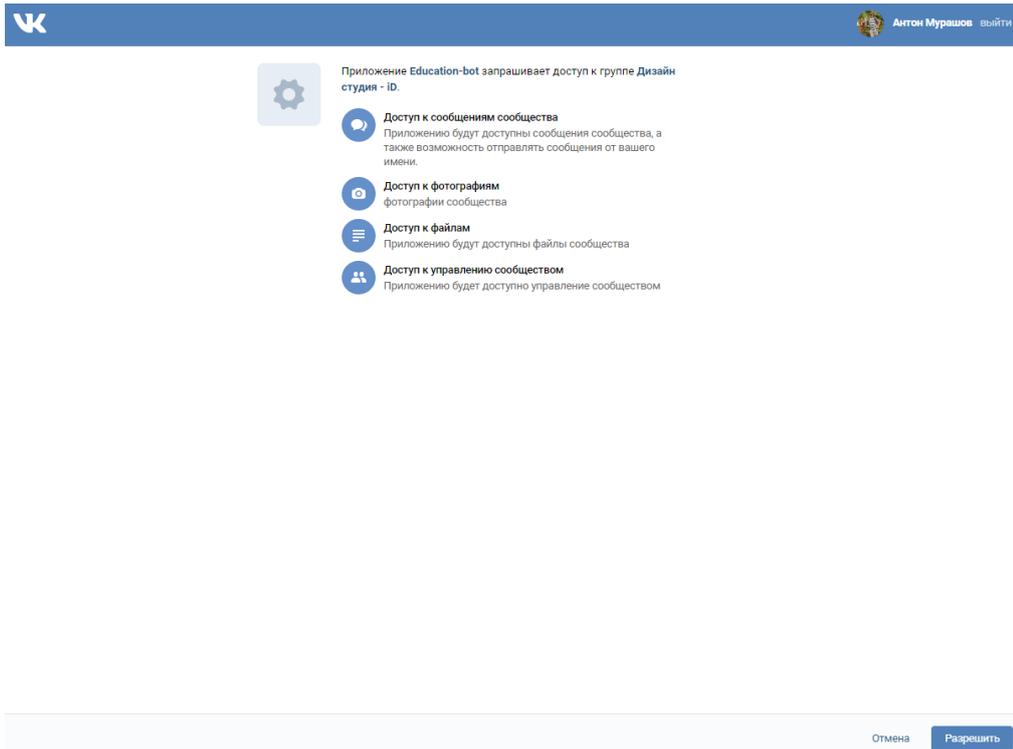


Рис. 2.19. Подтверждение авторизации приложения в группе

После подтверждения, браузер также будет перенаправлен на страницу, указанную в переменной окружения REDIRECT_URI после чего произойдет выполнение GET-запроса для авторизации группы. Пример вызова метода представлен на рис. 2.20.

```
function* vkAuthGroupWorker(action: {
  payload: {
    code?: string;
    group_id: string;
  };
}) {
  const { code, group_id } = action.payload;
  try {
    const response: AxiosResponse<{
      status: string;
      group_id: number;
    }> = yield call(() =>
      API.get('/auth-group', {
        params: { code, group_id },
      })
    );
    if (response.status === 200) {
      history.push(`/office/${response.data.group_id}`);
    }
  } catch (error) {
    console.log('vkAuthGroupWorker', error);
  }
}
```

Рис. 2.20. Выполнение запроса для авторизации группы

Данный запрос вызовет метод `auth-group`, который находится в серверном приложении, который выполнит GET-запрос по адресу `https://oauth.vk.com/access_token`, для получения ключа доступа сообщества. Также в этом методе происходит добавление сервера приложения в сообщество и устанавливаются необходимые настройки для этого приложения в сообществе. Для проверки правильности используемых настроек необходимо перейти в настройки сообщества и выбрать пункт «Работа с API». Необходимо проверить, что ключ для доступа к сообществу и сервер приложения были добавлены.

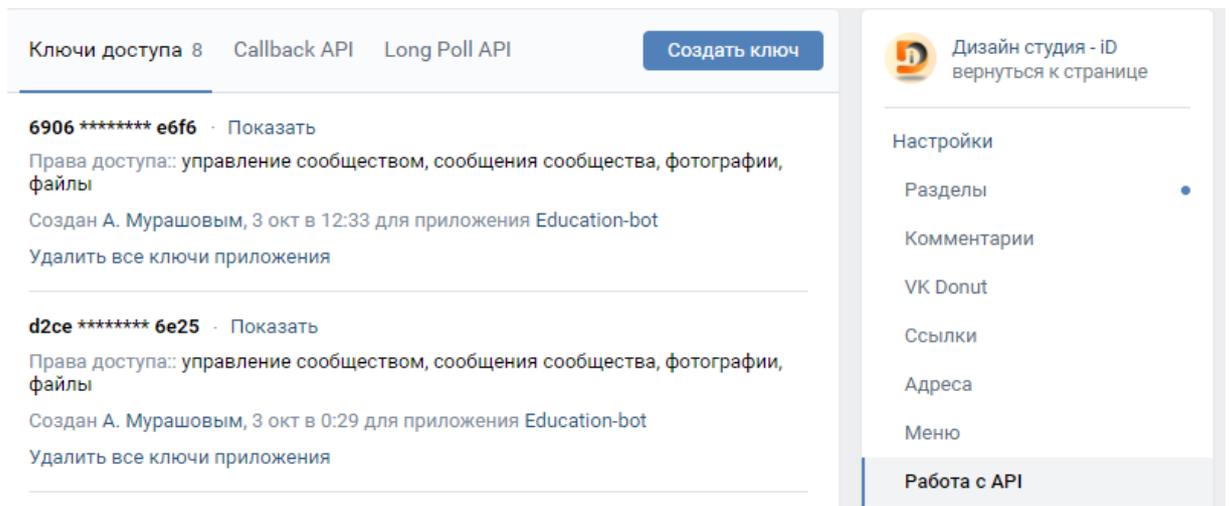


Рис. 2.21. Ключи доступа сообщества

Ключи доступа 8 Callback API Long Poll API **Добавить сервер**

Настройки сервера Типы событий Запросы

Название: EducationBot · Изменить

Версия API: 5.131

Адрес: ✓

Для получения уведомлений нужно подтвердить адрес сервера. На него будет отправлен **POST-запрос**, содержащий JSON:
{ "type": "confirmation", "group_id": 155725324 }

Строка, которую должен вернуть сервер: **a0708014**

Подтвердить

Секретный ключ: ?

Сохранить

Рис. 2.22. Добавленный сервер

После авторизации приложения в группе происходит перенаправление браузера пользователя по адресу `/office/:id`, где в качестве параметра `id` используется идентификатор группы. Так как адресная строка поменялась – происходит отрисовка компонента личного кабинета группы – `EditableGroupPage`. Как этот компонент выглядит в браузере, можно посмотреть на рис. 2.23.

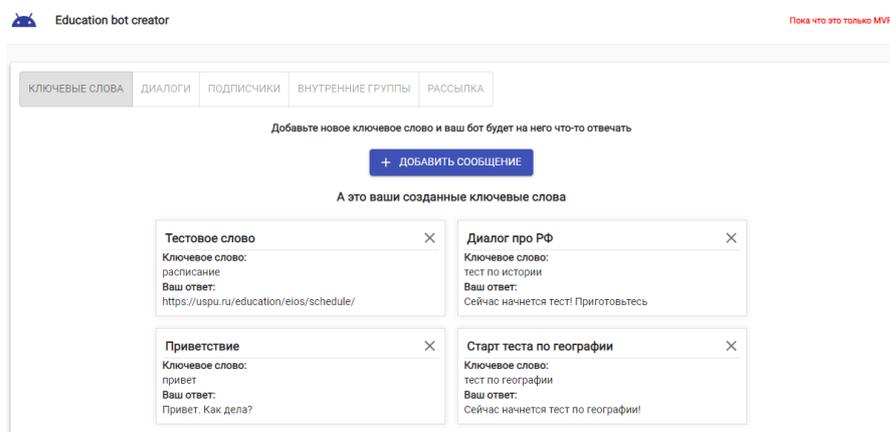


Рис. 2.23. Компонент `EditableGroupPage` в браузере

На данной странице представлен компонент ButtonGroup, который содержит набор кнопок, при переключении которых происходит отрисовка следующих компонентов:

```
const buttons = [
  {
    key: 'keyword',
    title: 'Ключевые слова',
    component: <NewKeyword />,
  },
  {
    key: 'dialog',
    title: 'Диалоги',
    component: <NewDialog />,
  },
  {
    key: 'subscribes',
    title: 'Подписчики',
    component: <Subscribes />,
  },
  {
    key: 'innerGroups',
    title: 'Внутренние группы',
    component: <InnerGroups />,
  },
  {
    key: 'mailing',
    title: 'Рассылка',
    component: <Mailing />,
  },
];
```

Рис. 2.24. Структура данных для компонента ButtonGroup

На рис. 2.24 видно, что для вкладки «Ключевые слова» используется компонент NewKeyword. Для вкладки «Диалоги» – NewDialog. Для вкладки «Подписчики» – Subscribes. Для вкладок «Внутренние группы» и «Рассылка» – InnerGroups и Mailing соответственно.

При выборе вкладки «Ключевые слова» пользователь может просматривать созданные ключевые слова, добавлять новые и удалять существующие. Добавленные ключевые слова распознаются в сообщении и пользователь, который написал сообщение в сообщество получит нужный ответ. Для добавления ключевого слова необходимо нажать кнопку «Добавить сообщение». Браузер пользователя будет перенаправлен по адресу /office/:id/new-message. В следствие чего будет отрисован компонент NewMessagePage. На рис 2.25 показано, как данный компонент выглядит в браузере.

Рис. 2.25. Компонент NewMessagePage

Перед добавлением ключевого слова пользователю необходимо дать название ключевого слова, написать само ключевое слово и ввести в поле «Ваше сообщение» ответ, который будет отправлен пользователю. Также пользователь может выбрать диалог, который будет начинаться после написания этого ключевого слова. Например, создадим слово «расписание». Ответом на это ключевое слово будет являться ссылка на расписание занятий. После нажатия на кнопку «Сохранить» будет выполнен PUT-запрос по адресу `/office/:id/keyword`. Данный запрос вызовет серверный метод для создания ключевого слова. Код этого метода представлен на рис 2.26.

```

app.put('/api/office/:id/keyword', async (req, res) => {
  const token = await getGroupToken(req.params.id);
  if (!token) {
    res.status(403).send();
    return;
  }
  await req.on('data', (d) => {
    const body = JSON.parse(d);
    const insertResult = {
      group_id: req.params.id,
      ...body.data,
      keyword: body.data.keyword.toLowerCase(),
      dialogId: body.data.dialogId ? ObjectId(body.data.dialogId) : null,
    };
    req.app.locals.keywordsCollection.insertOne(insertResult, (err, result) => {
      if (err) {
        return res.status(400).send();
      }
      return res.status(201).send();
    });
  });
});

```

Рис. 2.26. Метод добавления ключевого слова

Ключевое слово сохраняется в базе данных с привязкой к группе и диалогу, если таковой был указан. После чего браузеру пользователя

отправляется успешный код запроса 201, что говорит о том, что ключевое слово было успешно создано. После чего происходит перенаправление браузера пользователя на странице /office/:id, где будет выведено созданное ключевое слово.

Для того, чтобы серверное приложение успешно отвечало на написанные сообщения в сообщество был написан метод, который находится по адресу /api/office/:id. На данный адрес приходят POST-запросы от сервера VK с параметрами, указанными в таблице ниже.

Таблица 2.4

Параметры, отправляемые мессенджером VK на вебхук web-приложения «Education-bot-creator»

| Наименование параметра | Описание |
|------------------------|--|
| type | Тип события (для новых сообщений приходит message_new) |
| object | Объект, инициировавший событие |
| group_id | ID сообщества, в котором произошло событие |

Например, если пользователь вступил в сообщество, в приложение от сервера VK придет событие group_join следующего вида:

```
{ "type": "group_join", "object": { "user_id": 1, "join_type": "approved" }, "group_id": 1 }
```

Для проверки работоспособности необходимо отправить сообщение в сообщество с нужным ключевым словом. В ответ придет:

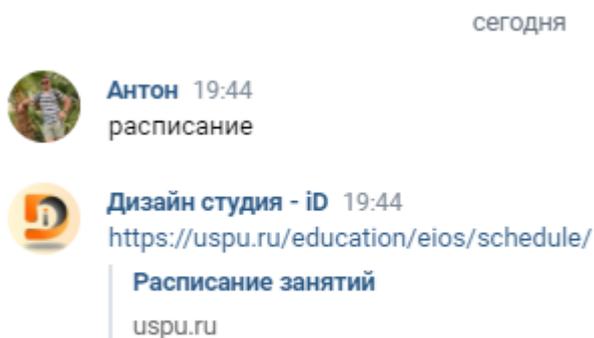


Рис. 2.27. Ответ пользователю

Следующим пунктом идет вкладка «Диалоги». Для этого существует компонент NewDialog, который выводит список существующих диалогов, позволяет удалять диалоги и создавать новый. Для создания диалогов используется компонент NewDialogPage, визуальная часть которого представлена на рис 2.28.

Новый диалог

Название диалога
Новое название диалога

Вопрос 1

Новый вопрос

| | | | | |
|-----------|---------------|---------|---|--|
| 1. Ответ: | Новый ответ | Баллов: | 0 | |
| 2. Ответ: | Новый ответ 2 | Баллов: | 1 | |

+ ОТВЕТ

Вопрос 2

Вопрос номер 2

| | | | | |
|-----------|-----|---------|---|--|
| 1. Ответ: | Да | Баллов: | 1 | |
| 2. Ответ: | Нет | Баллов: | 0 | |

+ ОТВЕТ

Вопрос 3

Вопрос последний

| | | | | |
|-----------|-----|---------|---|--|
| 1. Ответ: | Да | Баллов: | 1 | |
| 2. Ответ: | Нет | Баллов: | 0 | |

+ ОТВЕТ

ДОБАВИТЬ ВОПРОС

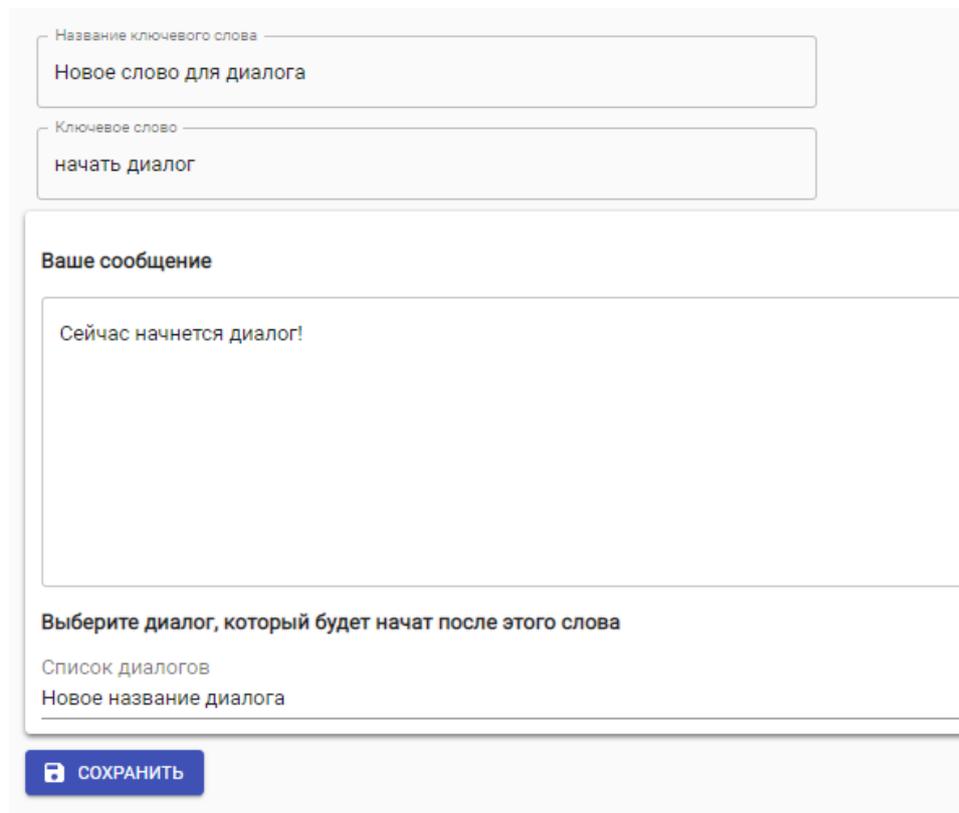
Пользователь может пройти диалог только один раз

СОХРАНИТЬ

Рис. 2.28. Компонент добавления нового диалога

Для добавления диалога необходимо дать ему название. После чего

добавить вопросы кнопкой «Добавить вопрос». При необходимости вопросу необходимо добавить ответы и количество баллов за ответы. Также можно поставить галочку единственности диалога, то есть можно ли будет начать данный диалог несколько раз. После нажатия на кнопку сохранить на серверное приложение отправляется PUT-запрос и вызывается метод добавления нового диалога в базу данных. После сохранения диалога, его необходимо привязать к ключевому слову. Для этого нужно добавить новое ключевое слово и указать необходимые параметры.



The screenshot shows a web form with the following elements:

- A text input field labeled "Название ключевого слова" (Keyword name) containing the text "Новое слово для диалога" (New word for dialog).
- A text input field labeled "Ключевое слово" (Keyword) containing the text "начать диалог" (start dialog).
- A section titled "Ваше сообщение" (Your message) containing a text area with the text "Сейчас начнется диалог!" (Dialog will start now!).
- A section titled "Выберите диалог, который будет начат после этого слова" (Select a dialog that will be started after this word) containing a dropdown menu with the text "Список диалогов" (List of dialogs) and "Новое название диалога" (New dialog name).
- A blue button labeled "СОХРАНИТЬ" (SAVE).

Рис. 2.29. Добавление нового ключевого слова с диалогом

После нажатия на кнопку «Сохранить» можно проверить корректность обрабатывания сообщений с диалогом. Для этого необходимо написать ключевое слово в сообщество, привязанное к диалогу.

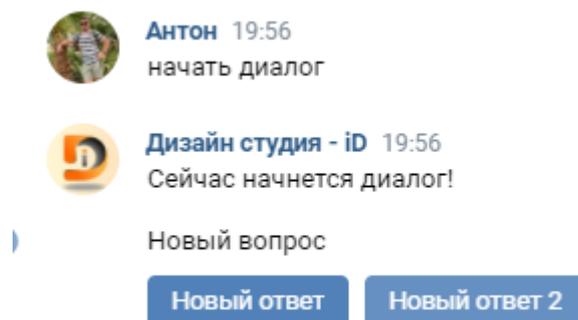


Рис. 2.30. Начало диалога

После нажатия на ответ пользователю будет отправлен следующий вопрос в диалоге.

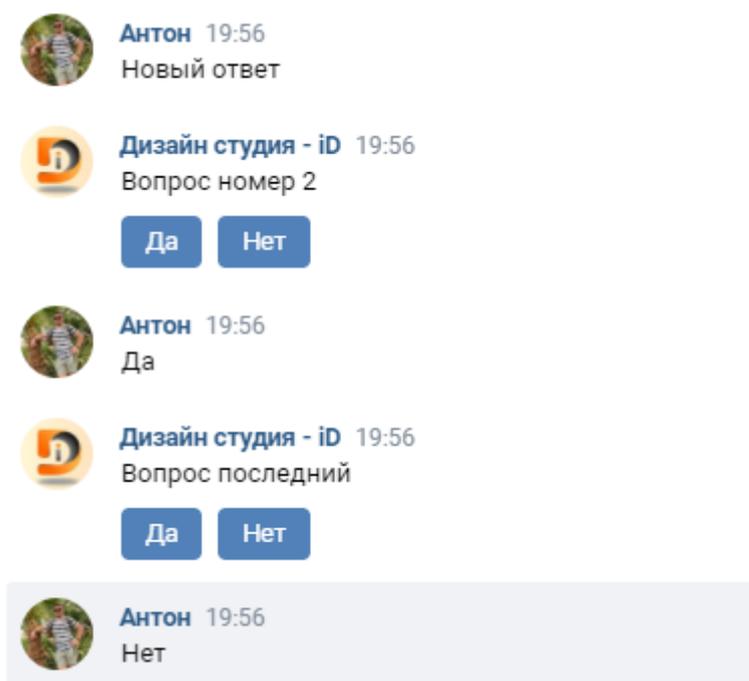


Рис. 2.31. Конец диалога

Для просмотра списка подписчиков сообщества используется вкладка «Подписчики». Для этого был реализован компонент Subscribes, при отрисовке которого отправляется GET-запрос для получения информации о количестве подписчиков. Визуальная часть представлена на рис. 2.32.

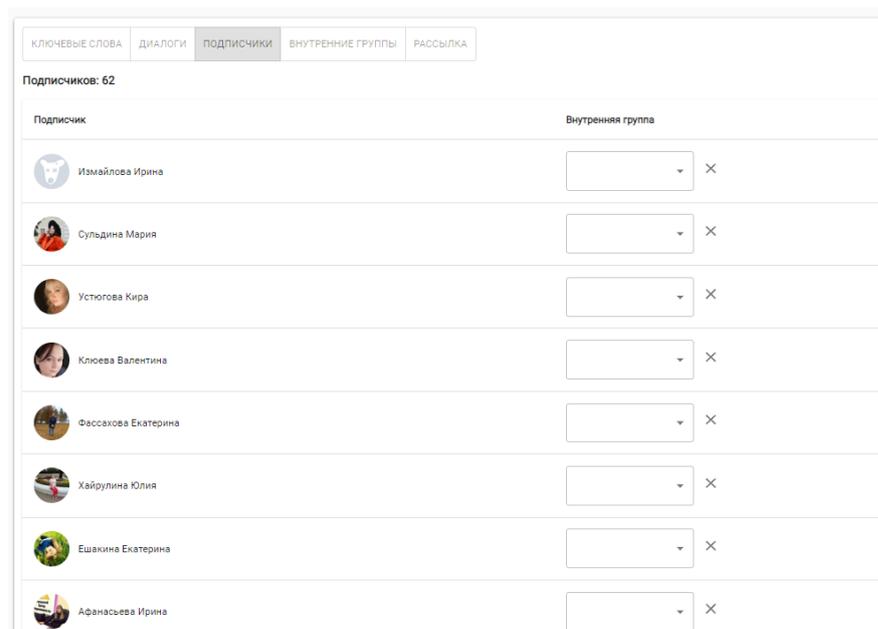


Рис. 2.32. Компонент Subscribes

По умолчанию идет запрос на получение 50 подписчиков. Если пользователь пролистает вниз, то увидит кнопку «Показать еще», которая отправляет следующий такой же запрос, но уже на получение следующего списка 50 подписчиков. Также компонент Subscribes позволяет объединять подписчиков в группы для создания рассылок. Для того, чтобы добавить подписчика во внутреннюю группу необходимо из выпадающего списка выбрать наименование нужной группы.

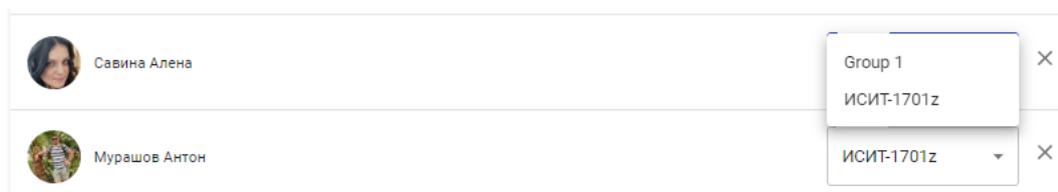


Рис. 2.33. Выпадающий список внутренних групп

Для удаления подписчика из внутренней группы необходимо нажать кнопку с крестиком.

При добавлении подписчика в группу на сервер приложений отправляется POST-запрос для выполнения метода `change-inner-group`, который используется добавления пользователя во внутреннюю группу, либо для изменения группы.

Вкладка «Внутренние группы» используется для добавления и удаления внутренних групп, который позволяют объединять подписчиков. Для вывода

списка групп используется компонент InnerGroups. Для добавления группы необходимо нажать кнопку «Создать группу», после чего браузер пользователя будет перенаправлен по адресу /office/:id/new-inner-group и будет отрисован компонент NewInnerGroupPage.

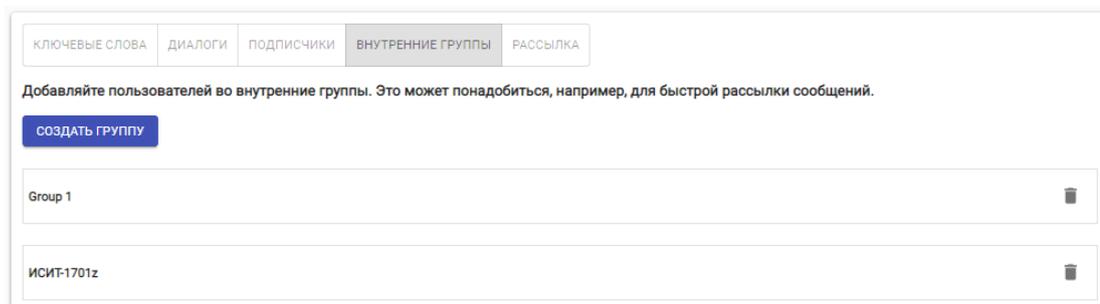


Рис. 2.34. Компонент InnerGroups

Форма добавления группы содержит единственное поле для ввода и кнопку «Сохранить». После ввода и нажатия на кнопку сохранить на сервер отправляется PUT-запрос.



Рис. 2.35. Компонент NewInnerGroupPage

При нажатии на кнопку «Крестик» на сервер отправляется DELETE-запрос, который в качестве параметра содержит ID-группы. На сервере вызывается метод для удаления внутренней группы из базы данных.

Еще одна полезная вкладка – «Рассылка». Она позволяет начать рассылку подписчикам, которые находятся во внутренних группах. Компонент рассылки называется Mail и содержит в себе одну кнопку, которая перенаправляет браузер пользователя на страницу /office/:id/new-mailing после чего отрисовывается компонент NewMailingPage.

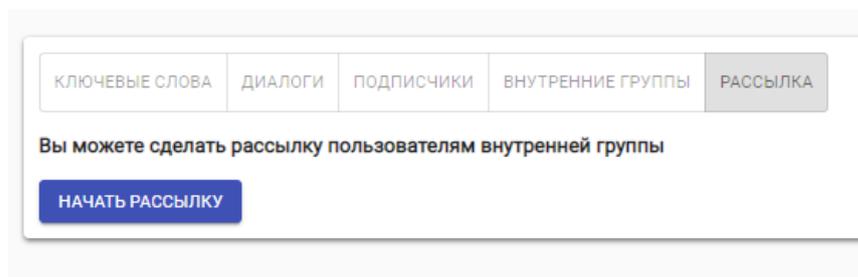


Рис. 2.36. Компонент Mailing

Рис. 2.37. Компонент NewMailingPage

Пользователь может выбрать одну или несколько групп. После чего необходимо ввести сообщение в поле для ввода. После того, как все эти действия будут проделаны необходимо нажать кнопку «Отправить».

Рис. 2.38. Заполненные поля в компоненте NewMailingPage

После отправки сообщения будет вызван POST-запрос, который возьмет ID-пользователей из базы данных, которые состоят в выбранной группе. После чего через API VK произойдет отправка сообщений выбранным пользователям. Результат данной работы продемонстрирован на рисунке 2.39 и 2.40.

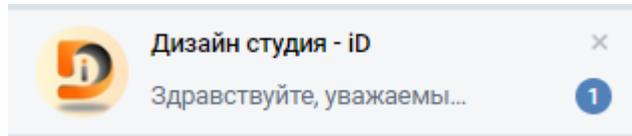


Рис. 2.39. Присланное сообщение от группы

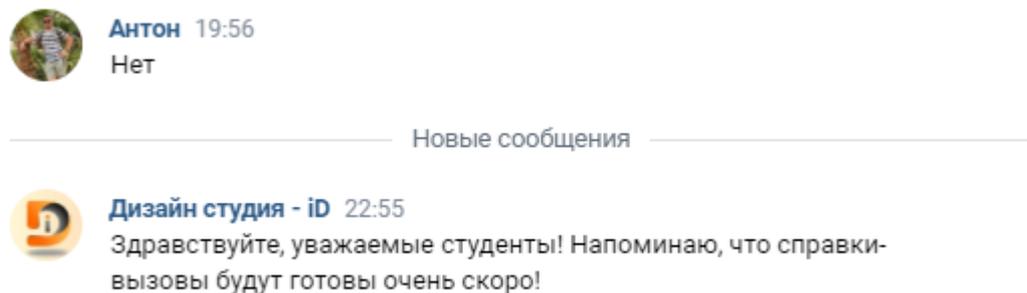


Рис. 2.40. Просмотр сообщения

Для работы с базой данных MongoDB необходимо использовать npm-пакет `mongodb`. Данная библиотека позволяет создать на сервере экземпляр класса для работы с облачной базой данных. Делается это через создание экземпляра класса `MongoClient` с указанием сервера, имени пользователя и пароля.

```
const mongoClient = new MongoClient(  
  'mongodb+srv://anton:password@cluster0.mjkd7.mongodb.net/myFirstDatabase?retryWrites=true&w=majority',  
  {  
    useUnifiedTopology: true,  
  }  
);  
let dbClient;
```

Рис. 2.41. Создание экземпляра класса MongoClient

После этого, посредством вызова метода `connect` у экземпляра класса в передаваемой `callback`-функции инициализируются коллекции, который находятся в переменной `app.locals`.

```

mongoClient.connect(function (err, client) {
  if (err) return console.log(err);
  dbClient = client;
  app.locals.botsCollection = client.db('educationBot').collection('bots');
  app.locals.keywordsCollection = client
    .db('educationBot')
    .collection('keywords');
  app.locals.dialogsCollection = client
    .db('educationBot')
    .collection('dialogs');

  app.locals.dialogsStarting = client
    .db('educationBot')
    .collection('dialogsStarting');

  app.locals.messagesHistory = client
    .db('educationBot')
    .collection('messagesHistory');

  app.locals.answersHistory = client
    .db('educationBot')
    .collection('answersHistory');

  app.locals.dialogsHistory = client
    .db('educationBot')
    .collection('dialogsHistory');

  app.locals.innerGroups = client.db('educationBot').collection('innerGroups');

  app.locals.usersInGroups = client
    .db('educationBot')
    .collection('usersInGroups');

  app.locals.currentUserToken = client
    .db('educationBot')
    .collection('currentUserToken');

  app.locals.currentGroupToken = client
    .db('educationBot')
    .collection('currentGroupToken');

  app.locals.mailings = client.db('educationBot').collection('mailings');

  app.listen(process.env.PORT || 8080, () => {
    console.log(`Сервер работает на порту ${process.env.PORT}`);
  });
});

```

Рис. 2.42. Инициализация коллекций БД

После того, как приложение было разработано и протестировано на машине разработчика, необходимо сделать так, чтобы это приложение работало одинаково везде – на всех платформах. Для этого используется Docker. Docker позволяет «упаковать» приложение в контейнер, который будет одинаково работать на всех операционных системах, где установлен Docker.

После установки docker в корневой папке приложения нужно создать Dockerfile и описать команды, необходимые для запуска приложения в контейнере. Для frontend-части приложения используются следующие команды

```
FROM node:12-alpine as build
WORKDIR /app
COPY ./package.json .
RUN npm install
COPY . .
RUN npm run build

FROM nginx:1.16.0-alpine
COPY --from=build /app/dist /usr/share/nginx/html
EXPOSE 80 443
ADD nginx/nginx.conf /etc/nginx/conf.d/default.conf
CMD ["nginx", "-g", "daemon off;"]
```

Рис. 2.43. Описание Dockerfile

В приведенном dockerfile директива FROM говорит Docker использовать образ NodeJS версии 12. Далее через WORKDIR устанавливается рабочая директория. С помощью директивы COPY копируется файл package.json из текущей директории в рабочую директорию /app в контейнере. После чего идет установка зависимостей команды npm install, копируются оставшиеся файлы из текущей директории в директорию /app в контейнере и запуск команды для сборки проекта npm run build.

После того, как приложение будет собрано в контейнере необходимо настроить веб-сервер. Для этого используется образ nginx версии 1.16. С помощью директивы COPY --from=build Docker понимает, что нужно скопировать файлы из папки /app/dist, которая была получена на предыдущем этапе сборки приложения, в папку /usr/share/nginx/html. Далее идет открытие портов 80 и 443, который контейнер будет слушать после запуска. Далее идет копирование данных файла nginx.conf из директории nginx в директорию /etc/nginx/conf.d в файл default.conf. Данный файл содержит настройки для веб-сервера nginx. После чего с помощью директивы CMD идет запуск веб-сервера nginx в режиме «демона» – то есть запущенном в фоновом режиме.

Файл nginx.conf представляет из себя настройки для веб-сервера nginx.

```

server {
    listen      80;
    server_name education-bot-creator.ru www.education-bot-creator.ru;
    location / {
        root    /usr/share/nginx/html;
        index   index.html index.htm;
        try_files $uri $uri/ /index.html;
    }
    location /api {
        proxy_pass 'http://127.0.0.1:8080';
    }
}

```

Рис. 2.44. Конфигурация веб-сервера nginx

В таблице ниже идет описание директив nginx.

Таблица 2.5

Описание директив web-сервера nginx

| Наименование директивы | Описание |
|------------------------|---|
| server | Задаёт конфигурацию для виртуального сервера |
| listen | Слушать 80 порт |
| server_name | Задаёт имя серверов |
| location/ | Задаёт правило, если пользователь перешел на один из server_name |
| root | Корневая папка с собранным приложением |
| index | Файл, который необходимо отправить браузеру пользователя |
| try_files | Проверяет существование файлов в заданном порядке |
| location /api | При обращении к server_name/api перенаправлять запрос на текущий сервер с портом 8080 |

Также необходимо создать Dockerfile для серверного приложения, которое отличается только тем, что копирует текущее приложение в контейнер и запускает его.

```

FROM node:12-alpine as builder
WORKDIR /app
COPY . .
RUN npm install
EXPOSE 8080
USER node
CMD ["node", "index.js"]

```

Рис. 2.44. Dockerfile серверного приложения

Далее необходимо зарегистрироваться на сайте <https://hub.docker.com> для отправки созданных образов в облачное хранилище. Для этого используются

команды `docker build` – для сборки, и `docker push` – для отправки образов.

После того, как образы были успешно отправлены, необходимо их скачать на удаленном сервере и запустить Docker-контейнеры. Для этого был использован облачный сервер VDS, в котором можно создать виртуальную машину Linux.

После того, как виртуальная система создана, необходимо через адресную строку подключиться к удаленному серверу и запустить Docker-контейнеры. Для этого необходимо подключиться по SSH к серверу, с помощью команды `docker pull` скачать образы. Далее, для запуска контейнеров используются команды `docker run -d -p 80:80 -net=host <id_образа frontend>` и `docker run -d -p 8080:8080 -net=host <id_образа backend>`. После выполнения этих команд приложение будет доступно по адресу `http://education-bot-creator.ru`.

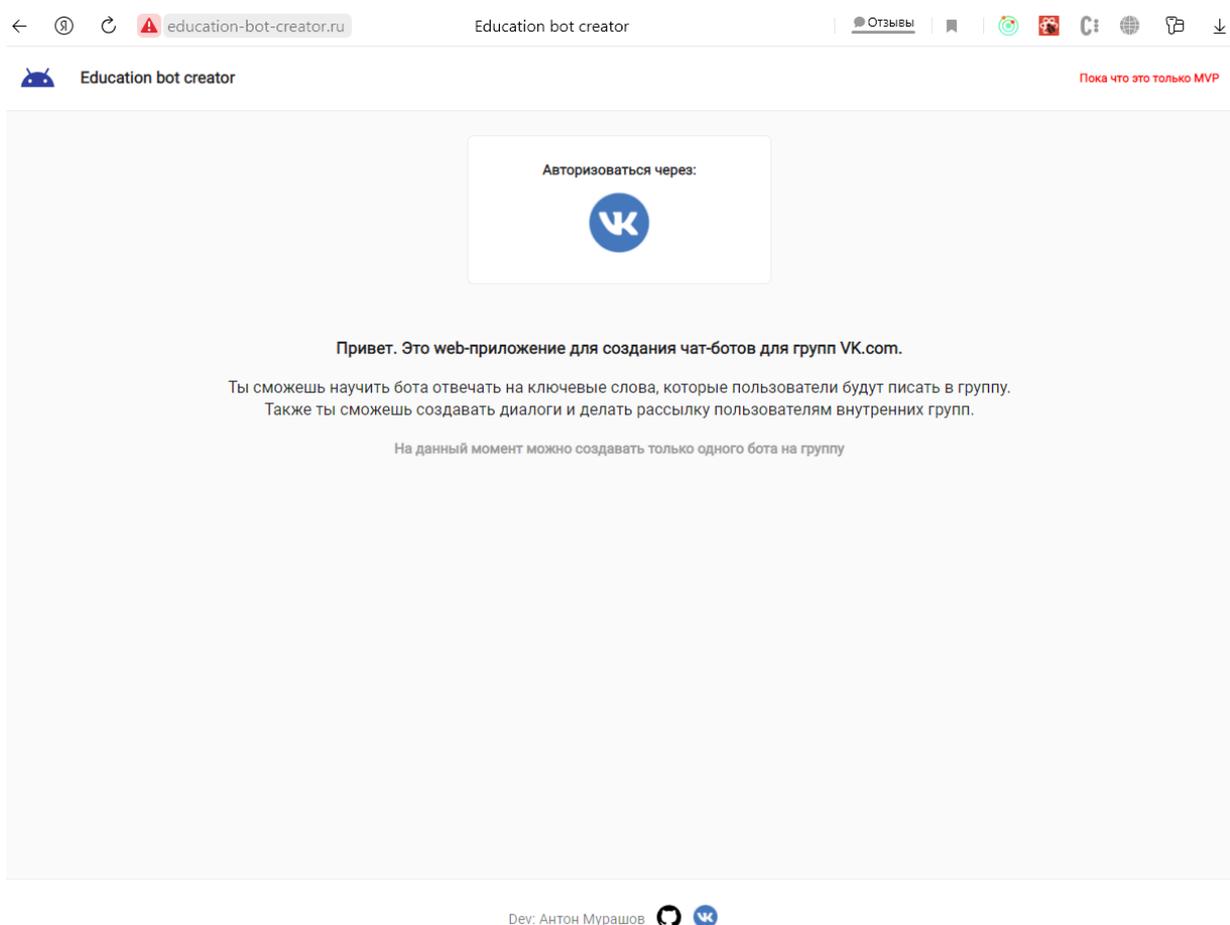


Рис. 2.46. Запущенное приложение на удаленном сервере.

2.4. Тестирование web-приложения «Education-bot-creator»

Проверка работы приложения «Education-bot-creator» была осуществлена несколькими тестировщиками – это студенты группы ИСИТ-1701z Уральского государственного педагогического университета. Данным студентам было предложено с использованием разработанного web-приложения и приложенного руководства пользователя создать чат-ботов и прикрепить их к группам и сообществам, в которых данные студенты являются администраторами.

Один из студентов выбрал группу «Маникюр» и добавил новое ключевое слово «Свободные места».

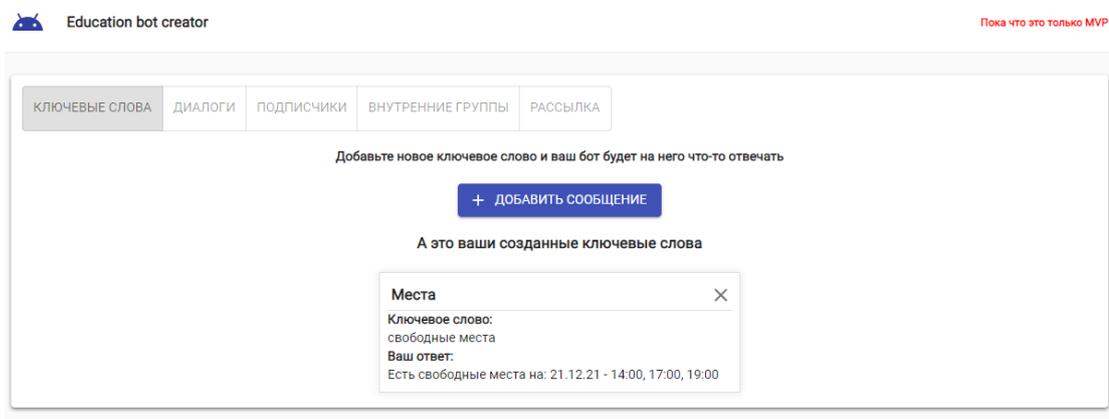


Рис. 2.47. Добавление ключевого слова.

После чего пользователь, написав ключевое слово в группу получил ответ, который им был ранее создан с помощью web-приложения «Education-bot-creator».

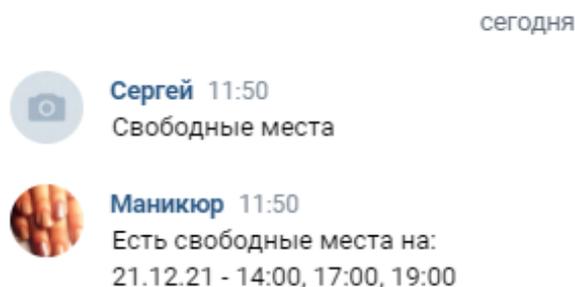


Рис. 2.48. Ответ группы на найденное ключевое слово.

Следующим этапом стало тестирование ведения диалогов с пользователем. Для этого был создан диалог в группе «Осаго онлайн», который

пользователи могли пройти и оценить качество работы сервиса «Осаго онлайн».

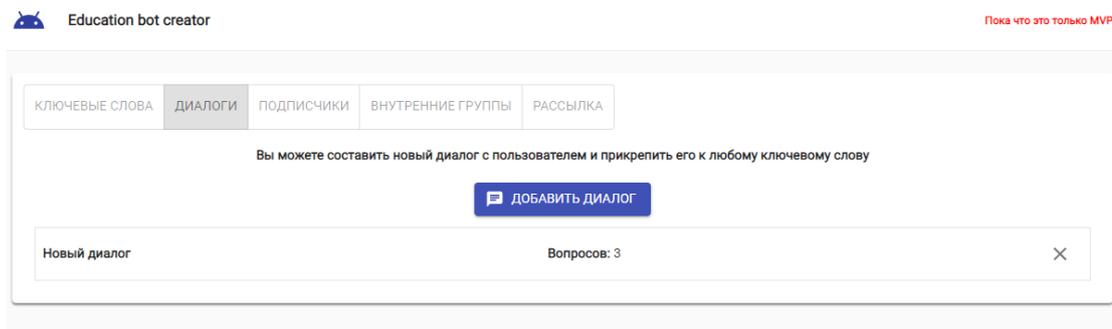


Рис. 2.49. Созданный диалог через web-приложение «Education-bot-creator».

Далее пользователь создал ключевое слово «оценка», после которого будет начинаться диалог.

После проделанных действий в группу «Осаго онлайн» было написано ключевое слово «оценка», после которого начался диалог, который был создан с помощью web-приложения «Education-bot-creator».

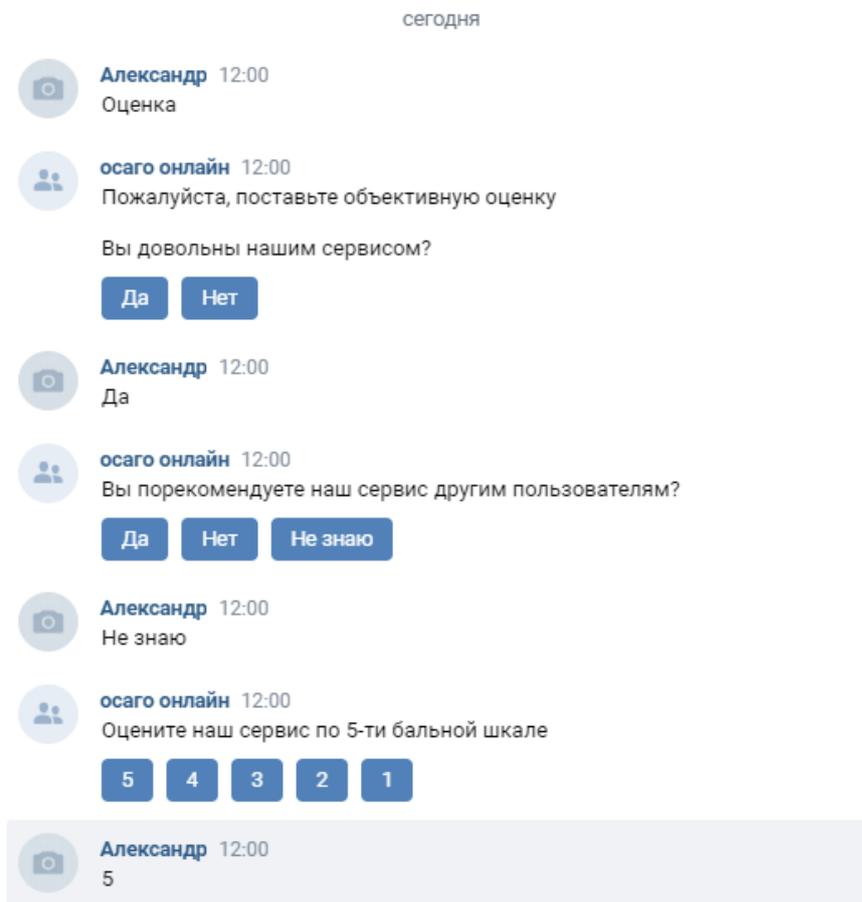


Рис. 2.50. Диалог с пользователем

Заключение

В рамках выпускной квалификационной работы было создано web-приложение, позволяющее пользователю автоматизировать работу преподавателей ВУЗов и администраторов различных сообществ в VK.

Был произведен анализ существующих технологий разработки web-приложений, исходя из аналитики и ссылаясь на техническое задание, выбраны такие технологии как ReactJS, NodeJS и MongoDB.

В ходе выполнения выпускной квалификационной работы были решены следующие задачи:

- Произведен анализ технологий для создания и использования образовательных чат-ботов, в котором было выяснено, что реализация SPA-архитектуры web-приложения отлично скажется на его производительности.

- Были проанализированы технологии, с помощью которых создаются образовательные чат-боты. Для реализации web-приложения «Education-bot-creator» использовались такие инструменты, как ReactJS, Docker, Nginx, MongoDB, платформа NodeJS и язык программирования JavaScript.

- В соответствие с техническим заданием было разработано web-приложение «Education-bot-creator».

- Произведено тестирование web-приложения «Education-bot-creator» и подготовлена сопроводительная документация.

Результатом работы является web-приложения для пользователей, которые являются администраторами сообществ в мессенджере VK.

Разработанное приложение удовлетворяет всем требованиям технического задания. Таким образом, следует считать, что задачи выпускной квалификационной работы полностью решены и цель достигнута.

Список информационных источников

1. Виртуальный собеседник, программа-собеседник, чат-бот // Wikipedia URL: https://ru.wikipedia.org/wiki/Виртуальный_собеседник (дата обращения: 01.04.2021).
2. Бот (программа), робот, интернет-бот // Wikipedia URL: [https://ru.wikipedia.org/wiki/Бот_\(программа\)](https://ru.wikipedia.org/wiki/Бот_(программа)) (дата обращения: 03.04.2021).
3. Чат-бот: как он работает и как сделать без программиста // Skillbox URL: <https://skillbox.ru/media/marketing/gayd-chatboty/> (дата обращения: 03.04.2021).
4. A Guide on Chatbots // Dzone URL: <https://dzone.com/articles/here-is-a-complete-guide-of-chatbots> (дата обращения: 03.04.2021).
5. Чат-бот для сайта: технологии создания, схемы, алгоритмы // Envybox URL: <https://envybox.io/blog/kak-rabotaet-chat-bot/#1> (дата обращения: 06.06.2021).
6. 3-minute Guide to Understanding « What is a Chatbot? » // Dzone URL: <https://dzone.com/articles/3-minute-guide-to-understand-what-is-a-chatbot> (дата обращения: 08.08.2021).
7. Web-приложение – определение // Wikipedia URL: <https://ru.wikipedia.org/wiki/Веб-приложение> (дата обращения: 12.08.2021).
8. Основные виды архитектур приложений // Stud24 URL: <https://www.stud24.ru/architecture/osnovnye-vidy-arhitektur-prilozhenij/264324-785939-page1.html> (дата обращения: 12.08.2021).
9. Язык гипертекстовой разметки HTML – определение // Wikipedia URL: <https://ru.wikipedia.org/wiki/HTML> – (дата обращения 03.09.2021).
10. Webhook – определение // Wikipedia URL: [https://ru.wikipedia.org/wiki/Webhook#:~:text=Вебхук%20\(англ.%20webhook\)%20в%20веб-разработке,hook\).%20Основной%20формат%20—%20JSON](https://ru.wikipedia.org/wiki/Webhook#:~:text=Вебхук%20(англ.%20webhook)%20в%20веб-разработке,hook).%20Основной%20формат%20—%20JSON) (дата обращение 24.10.2021).
11. Хэнчетт, Э. VueJS в действии // Э. Хэнчетт, Б. Листуон. – Санкт-Петербург: Питер, 2019. – 304 с. – ISBN 978-5-4461-1098-8

12. Spring Framework // Wikipedia URL: https://ru.wikipedia.org/wiki/Spring_Framework (дата обращения: 04.09.2021).
13. SQL и NoSQL: разбираемся в основных моделях баз данных // Tproger URL: <https://tproger.ru/translations/sql-nosql-database-models/> (дата обращения: 05.09.2021).
14. Установка и настройка Nginx // Selectel URL: <https://selectel.ru/blog/install-nginx/> (дата обращения: 08.09.2021).
15. Docker – определение // Wikipedia URL: <https://ru.wikipedia.org/wiki/Docker> (дата обращения: 14.09.2021).
16. ГОСТ 34.602-89. Информационная технология. Взамен ГОСТ 24.201-85; Введ. 1990-01-01. М. : Стандартиформ. 12 с. (Межгос. стандарт. Комплекс стандартов на автоматизированные системы. Техническое задание на создание автоматизированной системы).
17. Rest API – определение // Skillfactory URL: <https://blog.skillfactory.ru/glossary/rest-api/#:~:text=REST%20API%20—%20это%20способ,принято%20называть%20программным%20интерфейсом%20приложения> (дата обращения: 17.09.2021).
18. Как использовать чат-боты в бизнесе: 5 идей и 5 кейсов // VC URL: <https://vc.ru/services/93850-kak-ispolzovat-chat-boty-vbiznese-5-idey-i-5-keysov> (дата обращения: 18.09.2021).
19. Классификация и методы создания чат-бот приложений // Cyberleninka URL: <https://cyberleninka.ru/article/n/klassifikatsiya-i-metodysozdaniya-chat-bot-prilozheniy/viewer> (дата обращения: 18.09.2021).
20. Не соцсети, а чат-бот: зачем бизнесу мессенджеры // IGate URL: <https://igate.com.ua/news/24336-ne-sotsseti-a-chatbotzachem-biznesu-messendzhery> (дата обращения: 19.09.2021).
21. Примеры использования чат-ботов в бизнесе // VC URL: <https://vc.ru/flood/25197-business-bot> (дата обращения: 19.09.2021).
22. Использование чат-ботов для бизнеса // Webguru URL:

- https://webguru.pro/blog/use_chat_bots/ (дата обращения: 19.09.2021).
23. Почему именно мессенджеры // VC URL: <https://vc.ru/marketing/51778-kak-ispolzovat-bot-whatsapp-effektivnyusposob-generacii-kachestvennyh-lidov-cherez-messendzher> (дата обращения: 23.09.2021).
 24. Чем отличаются чат-боты в Telegram, WhatsApp, Viber, Facebook, VK // RPV Bot URL: <https://www.rpv-bot.ru/chem-otlichaetsyachat-bot-v-telegram-whatsapp-vk-viber-facebook> (дата обращения: 24.09.2021).
 25. React – JavaScript-библиотека для создания пользовательских интерфейсов: официальный сайт. – URL: <https://ru.react.js.org/> (дата обращения: 24.09.2021).
 26. React.js: понятное руководство для начинающих // Habr URL: <https://habr.com/ru/company/ruvds/blog/428077/> (дата обращения: 25.09.2021).
 27. Учебник: введение в React // Learn ReactJS URL: <https://learn-reactjs.ru/tutorial> (дата обращения: 26.09.2021).
 28. React Redux: официальный сайт. – URL: <https://react-redux.js.org/> (дата обращения: 26.09.2021).
 29. Введение в Redux & React-redux // Habr URL: <https://habr.com/ru/post/498860/> (дата обращения: 29.09.2021).
 30. Использование с React. Redux documentation in Russian // Gitbooks URL: <https://rajdee.gitbooks.io/redux-in-russian/content/docs/basics/UsageWithReact.html> (дата обращения: 04.10.2021).
 31. Redux-Saga: официальный сайт. – URL: <https://redux-saga.js.org/> (дата обращения: 04.10.2021).
 32. Разбираемся в redux-saga: От генераторов действий к сагам // Habr URL: <https://habr.com/ru/post/351168/> (дата обращения: 05.10.2021).
 33. Что такое npm? Общее Руководство для Начинающих // Hostinger URL: <https://www.hostinger.ru/rukovodstva/chto-takoe-npm> (дата обращения: 05.10.2021).
 34. Что такое npm? Гайд по Node Package Manager для начинающих // Proglib

- URL: <https://proglib.io/p/chto-takoe-npm-gayd-po-node-package-manager-dlya-nachinayushchih-2020-07-21> (дата обращения: 06.10.2021).
35. NodeJS: официальный сайт. – URL: <https://nodejs.org/en/> (дата обращения: 06.10.2021).
 36. Руководство по Node.js, часть 1: общие сведения и начало работы // Habr URL: <https://habr.com/ru/company/ruvds/blog/422893/> (дата обращения: 06.10.2021).
 37. Руководство по Node.js, часть 2: JavaScript, V8, некоторые приёмы разработки // Habr URL: <https://habr.com/ru/company/ruvds/blog/423153/> (дата обращения: 07.10.2021).
 38. Руководство по Node.js, часть 3: хостинг, REPL, работа с консолью, модули // Habr URL: <https://habr.com/ru/company/ruvds/blog/423701/> (дата обращения: 07.10.2021).
 39. Руководство по Node.js, часть 4: npm, файлы package.json и package-lock.json // Habr URL: <https://habr.com/ru/company/ruvds/blog/423703/> (дата обращения: 07.10.2021).
 40. Руководство по Node.js, часть 5: npm и npx // Habr URL: <https://habr.com/ru/company/ruvds/blog/423705/> (дата обращения: 08.10.2021).
 41. Руководство по Node.js, часть 6: цикл событий, стек вызовов, таймеры // Habr URL: <https://habr.com/ru/company/ruvds/blog/424553/> (дата обращения: 08.10.2021).
 42. Руководство по Node.js, часть 7: асинхронное программирование // Habr URL: <https://habr.com/ru/company/ruvds/blog/424555/> (дата обращения: 08.10.2021).
 43. MongoDB: the application data platform: официальный сайт. – URL: <https://www.mongodb.com/> (дата обращения: 10.10.2021).
 44. MongoDB Documentation: официальный сайт. – URL: <https://docs.mongodb.com/> (дата обращения: 12.10.2021).
 45. Основы MongoDB за 5 минут // Habr URL: <https://habr.com/ru/post/580760/>

- (дата обращения: 17.10.2021)..
46. MongoDB – определение // Wikipedia URL: <https://ru.wikipedia.org/wiki/MongoDB> (дата обращения: 17.10.2021).
 47. Полное практическое руководство по Docker: с нуля до кластера на AWS // Habr URL: <https://habr.com/ru/post/310460/> (дата обращения: 18.10.2021).
 48. Что такое Docker. URL: <https://docs.microsoft.com/ru-ru/dotnet/architecture/microservices/container-docker-introduction/docker-defined> (дата обращения: 21.10.2021).
 49. Docker самый простой и понятный tutorial. Изучаем докер, так, если бы он был игровой приставкой // Badcode URL: <https://badcode.ru/docker-tutorial-dlia-novichkov-rassmatrivaiem-docker-tak-iesli-by-on-byl-ighrovoi-pristavkoi/> (дата обращения: 21.10.2021).
 50. Nginx: официальный сайт. – URL: <https://nginx.org/ru/> (дата обращения: 22.10.2021).
 51. Nginx. Определение // Wikipedia URL: <https://ru.wikipedia.org/wiki/Nginx> (дата обращения: 22.10.2021).
 52. NGINX изнутри: рожден для производительности и масштабирования // Habr URL: <https://habr.com/ru/post/260065/> (дата обращения: 22.10.2021).
 53. Что такое Nginx // Eternalhost URL: <https://eternalhost.net/blog/sozдание-saytov/chto-takoe-nginx> (дата обращения: 22.10.2021).
 54. Подробная установка и настройка Nginx с примерами // Serveradmin URL: <https://serveradmin.ru/ustanovka-i-nastrojka-nginx/> (дата обращения: 22.10.2021).

Приложения

Приложение 1

Руководство пользователя

Составлен на основе ГОСТ «19.505-79 ЕСПД. Руководство оператора. Требования к содержанию и оформлению».

1. Введение

1.1. Область применения

Продукт предназначен для использования администраторами групп и сообществ в мессенджере VK.

1.2. Краткое описание возможностей

Web-приложение «Education-bot-creator» предоставляет следующие возможности:

- Создание ключевых слов и ответов пользователям.
- Создание диалогов с пользователем.
- Рассылка сообщений нескольким пользователям.

1.3. Уровень подготовки пользователя

Для управления системой пользователь должен иметь минимальный набор знаний и навыков работы с компьютером и браузером.

1.4. Перечень эксплуатационной документации

Настоящее «Руководство пользователя».

2. Назначение и условия применения

2.1. Назначение системы

Web-приложение «Education-bot-creator» предназначено для создания чат-ботов в группах и сообществах мессенджера VK.

2.2. Условия применения

Необходимо иметь выход к сети Интернет. В качестве устройства управления может выступать любой компьютер или ноутбук, с установленным браузером.

3. Подготовка к работе

3.1. Состав и содержание дистрибутивного носителя данных

Система не требует установки, управление происходит через любой веб-браузер.

3.2. Порядок загрузки данных и проверка работоспособности

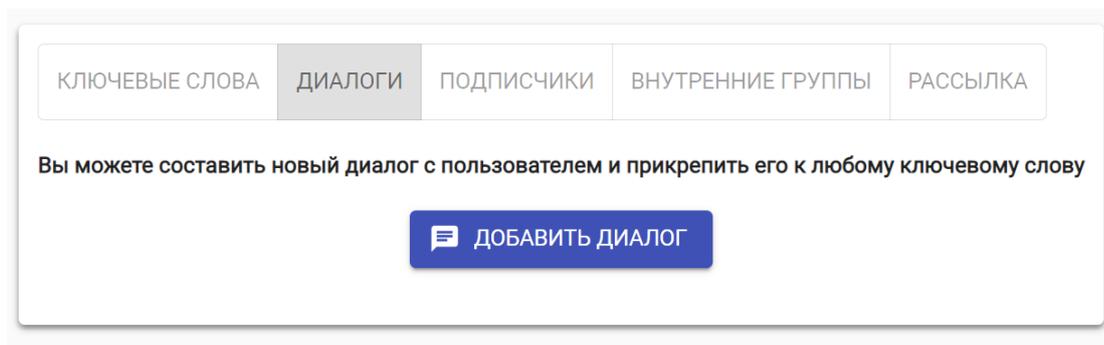
- 1) в браузере перейти по адресу <http://education-bot-creator.ru>;
- 2) проверить, что открылась страница с названием Education bot creator

4. Описание операций

Для пользователя доступны следующие операции:

- 1) авторизация пользователя в web-приложении с помощью мессенджера VK;
- 2) просмотр групп и сообществ, в которых пользователь является администратором;
- 3) добавление и удаление ключевых слов:

Для добавления диалога необходимо переключиться на вкладку «Диалоги» и нажать кнопку «Добавить диалог»



Заполнить форму для ввода и нажать кнопку «Сохранить»

Новый диалог

Название диалога

Вопрос 1

1. Ответ: _____ Баллов:0 

Пользователь может пройти диалог только один раз

- 4) просмотр списка пользователей, которые есть в выбранной группе или сообществе;
- 5) добавление и удаление внутренних групп;
- б) рассылка пользователям, которые были добавлены во внутреннюю группу.

5. АВАРИЙНЫЕ СИТУАЦИИ

Если веб-страница веб-приложения «Education-bot-creator» не загружается, необходимо:

- 1) перезагрузить устройство;
- 2) проверить наличие интернета;

6. РЕКОМЕНДАЦИИ ПО ОСВОЕНИЮ

Для успешного освоения системы изучить настоящее «Руководство пользователя» и выполнить действия, указанные в пункте 4.

Листинг backend-программы

```
async function messageSend(id, message, keyboard, token) {
  const randomId = Math.floor(Math.random() * 10000);

  const body = new URLSearchParams({
    // payload: JSON.stringify(payload),
    random_id: randomId,
    user_id: id,
    // peer_id: id,
    message,
    // v: '5.131',
  });

  console.log('messageSend', id, message, token);

  if (keyboard) {
    body.append('keyboard', keyboard);
  }

  fetch(
    `https://api.vk.com/method/messages.send?access_token=${token}&v=${V}`,
    {
      method: 'POST',
      headers: { 'Content-Type': 'application/x-www-form-urlencoded' },
      body: body.toString(),
    }
  )
  .then((res) => res.json())
  .then((d) => {
    console.log(d);
  })
  .then(console.log);
}
```

```

}

async function messagesSend(peerIds, message, token) {
  const randomId = Math.floor(Math.random() * 10000);

  const body = new URLSearchParams({
    random_id: randomId,
    peer_ids: peerIds,
    message,
  });

  console.log('messagesSend', peerIds, message, token);

  fetch(
    `https://api.vk.com/method/messages.send?access_token=${token}&v=${V}`,
    {
      method: 'POST',
      headers: { 'Content-Type': 'application/x-www-form-urlencoded' },
      body: body.toString(),
    }
  )
  .then((res) => res.json())
  .then((d) => {
    console.log(d);
  })
  .then(console.log);
}

function sleep(ms) {
  return new Promise((resolve) => setTimeout(resolve, ms));
}

function prepareKeyboard(question) {
  const buttons = question.answers.map((el) => ({

```

```

action: {
  type: 'text',
  // payload: `point:${el.point}/answerNum:${el.answerNum}/answer:${el.answer}`,
  payload: JSON.stringify({
    point: el.point,
    answerNum: el.answerNum,
    answer: el.answer,
    questionNum: question.questionNum,
  }),
  label: el.answer,
},
color: 'primary',
));
const keyboard = {
  buttons: [buttons],
  inline: true,
};

return JSON.stringify(keyboard);
}

function addMessageHistory(object) {
  app.locals.messagesHistory.insertOne(object);
}

function addDialogHistory(object) {
  app.locals.dialogsHistory.insertOne(object);
}

async function checkUserToken(user_id) {
  const field = await app.locals.currentUserToken.findOne({
    user_id: user_id.toString(),
  });
  if (!field) return false;
}

```

```
    return Date.now() <= field.date_end;
  }
```

```
async function getUserToken(user_id) {
  const res = await checkUserToken(user_id);
  if (!res) return null;
  const field = await app.locals.currentUserToken.findOne({
    user_id: user_id.toString(),
  });

  return field.token;
}
```

```
async function getGroupToken(group_id) {
  const field = await app.locals.currentGroupToken.findOne({
    group_id: group_id.toString(),
  });
  return field ? field.token : null;
}
```

```
app.use(express.urlencoded({ extended: true }));
app.get('/api/auth', async (req, res) => {
  const code = req.query.code;
  await https
    .get(
```

```
`https://oauth.vk.com/access_token?client_id=${CLIENT_ID}&client_secret=${CLIENT_SECRET}&redirect_uri=${REDIRECT_URI}&code=${code}`,
```

```
    (responseVk) => {
      responseVk.on('data', async (d) => {
        const hasError = JSON.parse(d);
        if (hasError.error) {
          res.status(404).send(hasError);
        }
        return;
      });
    }
  );
}
```

```

    }
    const data = JSON.parse(d);
    const currentUserToken = {
      user_id: data.user_id.toString(),
      token: data.access_token,
      expires_in: data.expires_in * 1000,
      date_record: Date.now(),
      date_end: Date.now() + data.expires_in * 1000,
    };
    await app.locals.currentUserToken.deleteOne({
      user_id: data.user_id.toString(),
    });
    await app.locals.currentUserToken.insertOne(currentUserToken);

    res.status(200).send({ user_id: data.user_id, email: data.email });
  });
}
)
.on('error', (e) => {});
});

/**
 * Получение информации о пользователе
 */
app.get('/api/user-info', async (req, res) => {
  const token = await getUserToken(req.cookies.user_id);
  if (token === null) {
    res.status(403).send();
    return;
  }
  await https.get(

```

```

`https://api.vk.com/method/users.get?user_ids=${req.cookies.user_id}&fields=bdate&access_token
=${token}&v=${V}`,
```

```

(resVk) => {
  resVk.on('data', (d) => {
    const data = JSON.parse(d);
    if (data.error) {
      res.status(404).send(data.error);
      return;
    }
    res.status(200).send(data.response[0]);
  });
}
);
});

```

```

app.get('/api/user-groups', async (req, res) => {
  const token = await getUserToken(req.cookies.user_id);
  if (token === null) {
    res.status(403).send();
    return;
  }
  await https.get(

```

`https://api.vk.com/method/groups.get?user_ids=\${req.cookies.user_id}&extended=1&filter=admin
&access_token=\${token}&v=5.131`,

```

(resVk) => {
  let rawData = "";
  resVk.on('data', (chunk) => {
    rawData += chunk;
  });
  resVk.on('end', () => {
    const data = JSON.parse(rawData);
    if (data.error) {
      res.status(404).send(data.error);
      return;
    }
  }

```

```

        res.status(200).send(data.response.items);
    });
}
);
});

/**
 * Авторизация группы
 * С использованием Callback API
 */
app.get('/api/auth-group', async (req, res) => {
    const code = req.query.code;
    let group_id = req.query.group_id;
    let token_group;

    console.log('group_id', group_id);
    const token = await getGroupToken(group_id);
    if (token) {
        res.status(200).send({
            status: 'success',
            group_id,
        });
        return;
    }
    https
        .get(
`https://oauth.vk.com/access_token?client_id=${CLIENT_ID}&client_secret=${CLIENT_SECRET}&redirect_uri=${REDIRECT_URI_GROUP}&code=${code}`,
        (responseVk) => {
            responseVk.on('data', async (d) => {
                const data = JSON.parse(d);
                if (data.error) {
                    res.status(404).send(data);
                }
            });
        });
    }
});

```

```

    return;
  }
  await app.locals.currentGroupToken.deleteOne({
    group_id: data.groups[0].group_id.toString(),
  });
  await app.locals.currentGroupToken.insertOne({
    group_id: data.groups[0].group_id.toString(),
    token: data.groups[0].access_token,
  });
  token_group = data.groups[0].access_token;
  group_id = data.groups[0].group_id;
  // Проверим есть ли сервер в группе уже
  https.get(

```

```

`https://api.vk.com/method/groups.getCallbackServers?group_id=${group_id}&access_token=${to
ken_group}&v=5.131`,

```

```

    (responseVk) => {
      responseVk.on('data', (d) => {
        const data = JSON.parse(d);
        const isFindServer = data.response.items.find(
          (el) => el.title === SERVER_NAME
        );
        console.log(isFindServer);
        if (isFindServer) {
          res.status(200).send({
            status: 'success',
            group_id,
          });
          return;
        }
        // Добавим сервер в группу
        https.get(

```

```

`https://api.vk.com/method/groups.addCallbackServer?group_id=${group_id}&url=${TEST_URL

```

```
}/api/office/${group_id}&secret_key=${secret_key_group}&title=${SERVER_NAME}&access_t  
oken=${token_group}&v=5.131`,
```

```
(responseVk) => {  
  responseVk.on('data', (d) => {  
    const data = JSON.parse(d);  
    if (secret_key_group === null) {  
      console.log(secret_key_group, 'null');  
      return;  
    }  
    if (data.error) {  
      console.log('error addCallbackServer', data.error);  
      res.status(403).send(data.error);  
      return;  
    }  
    const server_id = data.response.server_id;  
    // Установим настройки для сервера  
    https.get(  

```

```
`https://api.vk.com/method/groups.setCallbackSettings?group_id=${group_id}&server_id=${server  
_id}&access_token=${token_group}&v=5.131&message_new=1&group_join=1&api_version=${  
V}`,
```

```
(resVk) => {  
  resVk.on('data', (d) => {  
    const data = JSON.parse(d);  
    res.status(200).send({  
      status: 'success',  
      group_id,  
    });  
  });  
}  
);  
});  
}  
);
```

```

        });
    }
    );
    });
}
)
.on('error', (e) => {});
});

```

// Подтвердим адрес добавленного сервера

```

app.post('/api/office/:id', async (req, res) => {
  let confirmationCode = null;
  const token = await getGroupToken(req.params.id);
  if (!token) {
    res.status(403).send();
    return;
  }
  req.on('data', async (d) => {
    const body = JSON.parse(d);
    console.log(body);
    // Получим code для добавления сервера в группу
    if (body.type === 'confirmation') {
      https.get(
`https://api.vk.com/method/groups.getCallbackConfirmationCode?group_id=${req.params.id}&acc
ess_token=${token}&v=5.131`,
      (responseVk) => {
        responseVk.on('data', (d) => {
          const data = JSON.parse(d);
          if (data.error) {
            console.log('error getCallbackConfirmationCode', data.error);
            res.status(403).send(data.error);
          }
          return;
        });
      });
    }
  });
});

```

```

    }
    confirmationCode = data.response.code;
    res.send(confirmationCode);
  });
}
);
} else if (body.type === 'message_new') {
/**
 * Смотрим, есть ли открытый диалог
 */

const currentDialog = await app.locals.dialogsStarting.findOne({
  userId: body.object.message.peer_id,
});
// console.log('currentDialog', currentDialog);
if (currentDialog) {
  const payload = JSON.parse(body.object.message.payload);
  app.locals.answersHistory.insertOne({
    userId: body.object.message.from_id,
    dialogId: ObjectId(currentDialog.dialogId),
    dialogStartingId: ObjectId(currentDialog._id),
    ...payload,
  });
  // Ищем следующие вопросы в диалоге
  const questions = await app.locals.dialogsCollection.findOne({
    _id: ObjectId(currentDialog.dialogId),
  });
  // console.log('payload', payload);
  const nextQuestion = questions.questions.find(
    (el) => el.questionNum === payload.questionNum + 1
  );
  // console.log('nextQuestion', nextQuestion);
  if (nextQuestion) {
    const keyboard = prepareKeyboard(nextQuestion);

```

```

messageSend(
  body.object.message.from_id,
  nextQuestion.question,
  keyboard,
  token
);
addMessageHistory({
  user_id: body.object.message.from_id,
  dialogId: Object(currentDialog.dialogId),
  question: nextQuestion.question,
  date: new Date(),
});
} else {
  app.locals.dialogsStarting.deleteOne({
    _id: currentDialog._id,
  });
}
res.send('ok');
return;
}
/**
 * Сначала ищем совпадения по ключевым словам
 */
await req.app.locals.keywordsCollection.findOne(
  {
    group_id: body.group_id.toString(),
    keyword: body.object.message.text.toLowerCase(),
  },
  async (err, data) => {
    try {
      // #key:1
      if (err) {
        console.log('#key:1', err);
      }
      return;
    }
  }
);

```

```

}
console.log('keywordsCollection.findOne', token);

console.log(data);
messageSend(
  body.object.message.from_id,
  data.text,
  undefined,
  token
);
// #dlg:1
if (data.dialogId) {
  req.app.locals.dialogsCollection.findOne(
    {
      _id: ObjectId(data.dialogId),
    },
    async (err, innerData) => {
      if (err) {
        console.log('#dlg:1', err);
        return;
      }
      // Если диалог одноразовый
      if (innerData.isSingle) {
        const isOldDialog = await app.locals.dialogsHistory.findOne(
          {
            userId: body.object.message.from_id,
            dialogId: ObjectId(data.dialogId),
          }
        );
      }
      if (isOldDialog) {
        return;
      }
    }
  );
}
}

```

```

const insertObject = {
  userId: body.object.message.from_id,
  dialogId: ObjectId(data.dialogId),
};

req.app.locals.dialogsStarting.insertOne(insertObject);
addDialogHistory(insertObject);
// Если есть первое сообщение в диалоге, то отправим
if (innerData.questions[0]) {
  // сформировать сообщение и отправить пользователю
  const keyboard = prepareKeyboard(innerData.questions[0]);
  await sleep(2000);
  messageSend(
    body.object.message.from_id,
    innerData.questions[0].question,
    keyboard,
    token
  );
  addMessageHistory({
    user_id: body.object.message.from_id,
    dialogId: Object(data.dialogId),
    question: innerData.questions[0].question,
    date: new Date(),
  });
}
}
);
}
res.send('ok');
return;
} catch (error) {
  console.log(error);
  res.send('ok');
}

```

```

    }
  );
}
});
});

// Ключевые слова =====

app.get('/api/office/:id/keyword', async (req, res) => {
  const token = await getGroupToken(req.params.id);
  if (!token) {
    res.status(403).send();
    return;
  }
  const { id } = req.params;
  req.app.locals.keywordsCollection
    .find({
      group_id: id.toString(),
    })
    .toArray((err, items) => {
      if (err) {
        return res.status(400).send();
      }
      if (items.length === 0) {
        return res.status(204).send();
      }
      return res.status(200).send(items);
    });
});

app.put('/api/office/:id/keyword', async (req, res) => {
  const token = await getGroupToken(req.params.id);
  if (!token) {
    res.status(403).send();
    return;
  }

```

```

}
await req.on('data', (d) => {
  const body = JSON.parse(d);
  const insertResult = {
    group_id: req.params.id,
    ...body.data,
    keyword: body.data.keyword.toLowerCase(),
    dialogId: body.data.dialogId ? ObjectId(body.data.dialogId) : null,
  };
  req.app.locals.keywordsCollection.insertOne(insertResult, (err, result) => {
    if (err) {
      return res.status(400).send();
    }
    return res.status(201).send();
  });
});
});
});

app.delete('/api/office/:id/keyword', async (req, res) => {
  const token = await getToken(req.params.id);
  if (!token) {
    res.status(403).send();
    return;
  }
  await req.on('data', (d) => {
    const { id } = JSON.parse(d);
    try {
      req.app.locals.keywordsCollection.deleteOne({
        group_id: req.params.id.toString(),
        _id: ObjectId(id),
      });
    } catch {
      return res.status(400).send();
    }
    return res.status(200).send();
  });
});

```

```

    }
  });
});

// =====

// Диалоги =====

app.get('/api/office/:id/dialog', async (req, res) => {
  const token = await getGroupToken(req.params.id);
  if (!token) {
    res.status(403).send();
    return;
  }
  const { id } = req.params;
  req.app.locals.dialogsCollection
    .find({
      group_id: id.toString(),
    })
    .toArray((err, items) => {
      if (err) {
        return res.status(400).send();
      }
      if (items.length === 0) {
        return res.status(204).send();
      }
      return res.status(200).send(items);
    });
});

app.put('/api/office/:id/dialog', async (req, res) => {
  const token = await getGroupToken(req.params.id);
  if (!token) {
    res.status(403).send();
    return;
  }

```

```

}
await req.on('data', (d) => {
  const body = JSON.parse(d);
  const insertResult = {
    group_id: req.params.id,
    ...body,
  };
  req.app.locals.dialogsCollection.insertOne(insertResult, (err, result) => {
    if (err) {
      return res.status(400).send();
    }
    return res.status(201).send();
  });
});
});
});
app.delete('/api/office/:id/dialog', async (req, res) => {
  const token = await getGroupToken(req.params.id);
  if (!token) {
    res.status(403).send();
    return;
  }
  await req.on('data', (d) => {
    const { id } = JSON.parse(d);
    try {
      req.app.locals.dialogsCollection.deleteOne({
        group_id: req.params.id.toString(),
        _id: ObjectId(id),
      });
      return res.status(200).send();
    } catch {
      return res.status(400).send();
    }
  });
});
});

```

```

// =====
// Подписчики =====
app.get('/api/office/:id/subscribes', async (req, res) => {
  const { id } = req.params;
  const { offset } = req.query;
  const token = await getGroupToken(id);
  if (!token) {
    res.status(403).send();
    return;
  }
  https.get(
`https://api.vk.com/method/groups.getMembers?group_id=${id}&access_token=${token}&v=5.13
1&fields=photo_50&count=50&offset=${Number(
  offset
)}`,
  (resVk) => {
    let rawData = "";
    resVk
      .on('data', (chunk) => {
        rawData += chunk;
      })
      .on('end', () => {
        const data = JSON.parse(rawData).response;
        const newData = data.items.map(async (el) => {
          const userInGroup = await app.locals.usersInGroups.findOne({
            user_id: el.id,
            group_id: id,
          });
          if (userInGroup === null) {
            return {
              ...el,
              inner_group: null,
            };
          }
        });
      });
  });
}

```

```

    }
    const innerGroup = await app.locals.innerGroups.findOne({
      _id: ObjectId(userInGroup.inner_group_id),
    });
    return {
      ...el,
      inner_group: innerGroup._id,
    };
  });
  Promise.all(newData).then((d) => {
    res.status(200).send({ count: data.count, items: d });
  });
});
}
);
});
// =====

// Внутренние группы =====

app.get('/api/office/:id/inner-groups', async (req, res) => {
  const token = await getGroupToken(req.params.id);
  if (!token) {
    res.status(403).send();
    return;
  }
  const { id } = req.params;
  const data = await req.app.locals.innerGroups
    .find({
      group_id: id.toString(),
    })
    .toArray();
  res.status(200).send(data);
});

```

```

app.put('/api/office/:id/inner-groups', async (req, res) => {
  const token = await getGroupToken(req.params.id);
  if (!token) {
    res.status(403).send();
    return;
  }
  await req.on('data', (d) => {
    const body = JSON.parse(d);
    const insertResult = {
      group_id: req.params.id,
      ...body,
    };
    app.locals.innerGroups.insertOne(insertResult, (err, result) => {
      if (err) {
        return res.status(400).send();
      }
      return res.status(201).send();
    });
  });
});

```

```

app.delete('/api/office/:id/inner-groups', async (req, res) => {
  const token = await getGroupToken(req.params.id);
  if (!token) {
    res.status(403).send();
    return;
  }
  await req.on('data', (d) => {
    const { id } = JSON.parse(d);
    try {
      app.locals.innerGroups.deleteOne({
        group_id: req.params.id.toString(),
        _id: ObjectId(id),
      });
    } catch (err) {
      return res.status(400).send();
    }
  });
});

```

```

    });
    return res.status(200).send();
  } catch {
    return res.status(400).send();
  }
});
});

app.post('/api/office/:id/change-inner-group', async (req, res) => {
  const token = await getGroupToken(req.params.id);
  if (!token) {
    res.status(403).send();
    return;
  }
  await req.on('data', (d) => {
    const { id } = req.params;
    const { inner_group_id, user_id } = JSON.parse(d);
    try {
      app.locals.usersInGroups.insertOne({
        group_id: id,
        inner_group_id: ObjectId(inner_group_id),
        user_id: user_id,
      });
      return res.status(201).send();
    } catch {
      return res.status(400).send();
    }
  });
});

app.delete('/api/office/:id/change-inner-group', async (req, res) => {
  const token = await getGroupToken(req.params.id);
  if (!token) {
    res.status(403).send();

```

```

    return;
  }
  await req.on('data', async (d) => {
    const { id } = req.params;
    const { user_id } = JSON.parse(d);
    try {
      await app.locals.usersInGroups.deleteOne({
        group_id: id.toString(),
        user_id,
      });
      return res.status(200).send();
    } catch (er) {
      console.log(er);
      return res.status(404).send();
    }
  });
});

// =====

//=====PACЫЫJKA=====

app.get('/api/office/:id/mailling', async (req, res) => {
  const token = await getGroupToken(req.params.id);
  if (!token) {
    res.status(403).send();
    return;
  }
  const mailings = [];
  const innerGroups = [];
  const result = [];
  const cursor = await app.locals.mailings.find({
    group_id: req.params.id.toString(),
  });
});

```

```

const cursorInnerGroups = await app.locals.innerGroups.find();
await cursorInnerGroups.forEach((el) => innerGroups.push(el));

// Обогащаем массив данными
await cursor.forEach((el) => {
  const arrNames = el.groups.map((e) => {
    innerGroups.find((e1) => {
      console.log(ObjectId(e1._id) === ObjectId(e));
    });
  });
  // console.log(arrNames);
  mailings.push(el);
});
// console.log(mailings);
// return res.status(200).send(mailings);
});
app.post('/api/office/:id/ mailing', async (req, res) => {
  const token = await getGroupToken(req.params.id);
  if (!token) {
    res.status(403).send();
    return;
  }
  const { id } = req.params;
  req.on('data', async (d) => {
    const { group_id, groups, message } = JSON.parse(d);
    // Зафиксируем рассылку в БД
    const obj = {
      group_id,
      groups,
      message,
      date: new Date(),
    };
    await app.locals.mailings.insertOne(obj);
    const objectIdGroups = groups.map((el) => ObjectId(el));
  });
});

```

```
const usersIdsInGroups = [];
const cursor = await app.locals.usersInGroups.find({
  inner_group_id: { $in: objectIdGroups },
});
// Обогатим массив данными
await cursor.forEach((el) => usersIdsInGroups.push(el.user_id));
messagesSend(usersIdsInGroups, message, token);
res.status(200).send();
});
});

//=====

process.on('SIGINT', () => {
  dbClient.close();
  process.exit();
});
```