МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«ВЯТСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
ИНСТИТУТ МАТЕМАТИКИ И ИНФОРМАЦИОННЫХ СИСТЕМ
ФАКУЛЬТЕТ АВТОМАТИКИ И ВЫЧИСЛИТЕЛЬНОЙ ТЕХНИКИ
КАФЕДРА ЭЛЕКТРОННЫХ ВЫЧИСЛИТЕЛЬНЫХ МАШИН

Направление 09.03.01 - Информатика и вычислительная техника
(код и наименование направления)
Профиль – Программное и аппаратное обеспечение вычислительной техники

Допускаю к защите
Заведующий кафедрой ЭВМ
_____ / Долженкова М. Л. /
(подпись) *(Ф.И.О.)*

Разработка конструктора Telegram-ботов. Часть 1

Пояснительная записка выпускной квалификационной работы
ТПЖА 09.03.01.514 ПЗ

Разработал: студент гр.ИВТб-4301-04-00 _____ / Бушков Д. А. / _____

Руководитель:
к.т.н., доцент, зав. кафедрой ЭВМ _____ / Долженкова М. Л. / _____

Консультант: преподаватель кафедры ЭВМ _____ / Кошкин О. В. / _____

Нормоконтролер: к.т.н., доцент _____ / Скворцов А. А. / _____
(подпись) *(Ф.И.О)* *(дата)*

Киров 2024

Реферат

Бушков Д. А. Разработка конструктора Telegram-ботов. Часть 1: ТПЖА 09.03.01.514 ПЗ ВКР / ВятГУ, каф. ЭВМ; рук. Долженкова М. Л.– Киров, 2024. – Гр.ч. 8л. ф.А1; ПЗ 99 с., 50 рис., 3 табл., 14 форм., 12 источников, 4 прил.

КОНСТРУКТОР, TELEGRAM-БОТ, КЛИЕНТСКАЯ ЧАСТЬ, ВИЗУАЛЬНЫЙ РЕДАКТОР, СЕРВЕРНАЯ ЧАСТЬ, HTTP, GOLAND, POSTGRESQL, TYPESCRIPT, SOLID.JS, JSON, HTML, CSS.

Объект выпускной квалификационной работы - программное средство для упрощения создания и управления ботами в мессенджере Telegram.

Целью данной выпускной квалификационной работы является повышение скорости разработки, настройки и управления ботами, что позволит пользователям без специальных навыков программирования создавать эффективных ботов для различных целей.

Результат работы - конструктор Telegram-ботов, который будет предоставлять набор инструментов и функций для создания и настройки ботов, а также предоставлять возможности для их управления.

Содержание

Введение

В современном мире стали популярными такие приложения для быстрого общения как мессенджеры. Таких приложений достаточно много, но большинство пользователей сети интернет все чаще отдают предпочтение мессенджеру Telegram как наиболее удобному и надежному.

У Telegram имеется удобное API для создания ботов. Бот способен выполнять определенные команды, заданные пользователем через интерфейс Telegram. Данный функционал вполне может удовлетворять потребности компании в предоставлении разных услуг во многих сферах: от спортивных залов до доставки различных товаров.

Для создания ботов требуются высококвалифицированные программисты, которые обладают знаниями в одном или нескольких языках программирования. Наём таких сотрудников сопровождается довольно большими тратами для бизнеса. Также написание ботов с нуля - затратный по времени процесс.

Конструкторы или No-Code решения предоставляют возможность создавать приложения с использование визуальных блоков, что значительно упрощает процесс разработки. Эти инструменты предлагают графические интерфейсы, шаблоны и компоненты, которые облегчают создание приложений без необходимости писать много кода. Данные решения также существуют и в сфере Telegram-ботов: они предоставляют функции создания, редактирования и управления ботами.

К сожалению, большинство таких конструкторов предоставляют ограниченный функционал при бесплатном использовании, а также имеют закрытые способы хранения данных клиентов. Поэтому было принято решение выполнить анализ и разработать конструктор Telegram-ботов без данных недостатков.

1 Анализ предметной области

На данном этапе работы необходимо рассмотреть функции конструкторов Telegram-ботов, провести обзор существующих на данный момент аналогов, рассмотреть их возможности, выявить их недостатки и обосновать актуальность разработки нового конструктора.

1.1 Мессенджеры и конструкторы ботов

В нынешнее время становятся популярными такие приложения, как мессенджеры. Они позволяют быстро обмениваться сообщениями, а также предоставляют другие более полезные функции: обмен голосовых и видео сообщений, передача файлов, создание групп и т.д.

Наиболее популярными мессенджерами являются Telegram и WhatsApp. Данные мессенджеры предоставляют похожие функции, но Telegram предлагает больше функций безопасности и гибкости [1].

Большинство мессенджеров предлагают инструменты для создания чат-ботов, которые помогают пользователям выполнять типичные рутинные действия в автоматизированном режиме, значительно упрощая им жизнь. Для владельцев же самих ботов они стали незаменимыми помощниками в работе.

Чат-боты имеют множество плюсов, таких как:

– круглосуточный доступ;
– моментальный ответ на запрос пользователя;
– удобство использования, интуитивно понятный интерфейс;
– не требуется установка дополнительных программ, общение с ботом ведется через мессенджер.

Чат-бот используют в коммерческой деятельности для следующих сфер и задач:

– развлечения;
– поиск и обмен файлами;
– предоставление новостей;

– утилиты и инструменты;
– интеграция с другими сервисами;
– осуществление онлайн-платежей.

С популярностью ботов стали появляться все больше различных конструкторов, которые позволяют без наличия специальных знаний и навыков создать своего бота всего в несколько кликов.

Конструктором называется No-Code инструмент, который предназначен для быстрого создания ботов без знания каких-либо языков программирования. Иными словами, весь процесс создания – это нажатие тех или иных кнопок и ввода текста (например, название кнопки, текст сообщения и т.д.) [2].

Первое предназначение – упрощение работы. Ведь не все обладают глубокими знаниями и навыками программирования. Когда боты только появились, их могли разрабатывать только программисты, обладающие соответствующим опытом и навыками.

Помимо того, что конструкторы позволяют расширить аудиторию, способную создавать чат-ботов, они экономят время разработчикам. При наличии конструктора нет необходимости разрабатывать каждый раз отдельное приложение для выполнения типовых задач, так как конструктор предоставляет необходимый набор инструментов для быстрого создания бота без необходимости писать код.

Также некоторые конструкторы способны создавать ботов для разных мессенджеров, благодаря общим возможностям чат-ботов. Это достигается за счёт абстракций и различной реализации интерфейсов компонентов.

Но у конструкторов есть некоторые ограничения, например, при их использовании нельзя выйти за рамки возможностей самого конструктора, а также при выходе нового функционала в ботах мессенджера его реализация в конструкторе происходит с некоторой задержкой. Кроме того, боты, реализованные с помощью No-Code решения обычно менее производительные, чем их аналоги, написанные языке программирования.

1.2 Обзор аналогов

В подпунктах данного раздела рассматриваются существующие аналоги. В качестве рассматриваемых аналогов были выбраны приложения, реализующие функционал создания Telegram ботов с помощью конструктора.

1.2.1 Бот-платформа «ManyBot»

Один из наиболее популярных конструкторов. Работает внутри мессенджера Telegram. Он бесплатный и прост в использовании. Бот предоставляет создание бота с такими возможностями:

- отправка сообщений;
- создание меню;
- автопостинг из VK, Twitter, YouTube;
- поддержка нескольких языков.

Минусы конструктора состоят в том, что нет администрирования бота за пределами мессенджера, наличие рекламного сообщения в созданном боте, малое количество компонентов и их модификаций, а также отсутствие статистических данных по созданному боту.

1.2.2 Конструктор Telegram ботов «Puzzlebot»

Данный конструктор имеет намного больше возможностей, чем предыдущий сервис: удобный личный кабинет, интуитивный интерфейс, имеет намного больше компонентов, позволяющих реализовывать сложных ботов.

Минусы данного решения в том, что на бесплатном тарифе можно создать лишь одного бота и настроить до 15 команд, а также количество участников бота ограничено 150 пользователями.

1.2.3 Конструктор ботов «Botmother»

Довольно мощный сервис по созданию ботов, который имеет удобный интерфейс для администрирования и создания ботов, предоставляет много различных компонентов, возможность просмотра статистики созданного бота.

В бесплатном тарифе предоставляет лишь создание 10 тестовых ботов с ограниченным функционалом.

1.2.4 Сравнение аналогов

В таблице 1 представлено сравнение вышеперечисленных аналогов.

Таблица 1 – Сравнение аналогов

Критерии \ Аналоги	Manybot	Puzzlebot	Botmother
Удобный доступ для администрирования бота	нет	да	да
Изменение порядка вызовов компонентов	нет	да	да
Нет ограничений на использования компонентов	да	да	нет
Отсутствие рекламы	нет	нет	да

Как видно из таблицы 1 существующие решения имеют ряд недостатков. Также конструкторы больше ориентированы на получение прибыли и ограничивают функционал для бесплатного использования.

Учитывая недостатки рассмотренных аналогов разрабатываемое приложение должно обеспечивать следующий функционал:

- возможность создания ботов с помощью конструктора;
- возможность администрирования бота;
- возможность изменения порядка вызовов компонентов;
- отсутствие ограничений на использование компонентов;
- отсутствие рекламы.

1.3 Актуальность разработки

Telegram боты являются функциональными инструментами для многих пользователей, однако для их разработки зачастую требуются навыки программирования, что усложняет их внедрение в бизнес-процессы компаний. Конструкторы Telegram-ботов по большей части решают данную проблему, предоставляя пользователям удобный интерфейс для создания ботов под конкретные задачи.

Большинство компаний предоставляют ограниченный функционал конструктора, а для расширения их возможности требуют дополнительную плату, что

не всегда выгодно для конечного пользователя. Поэтому было принято решение о создании нового конструктора, который исключает вышеперечисленные недостатки, предоставляя пользователям свободную платформу для создания ботов.

1.4 Расширенное техническое задание

В данном подразделе представлено техническое задание на разработку конструктора Telegram-ботов.

1.4.1 Краткая характеристика области применения

Программа предназначена для создания Telegram-ботов, которые будут удовлетворять потребности клиентов в создании их бизнес-решений.

1.4.2 Основание для разработки

Функциональным назначением программы является предоставление клиентам возможности создания Telegram-ботов при помощи визуального редактора и без глубоких знаний языков программирования.

Программа должна эксплуатироваться на серверах клиента. Для использования конечному пользователю предъявляются требования знания процесса развёртывания серверных приложений.

1.4.3 Требования к структуре

Конструктор должен состоять из двух частей: серверной и клиентской. Серверная часть должна обеспечивать основной функционал конструктора. Клиентская часть предоставляет собой удобный пользовательский интерфейс.

1.4.4 Требования к серверной части конструктора

В подпунктах данного подраздела описываются требования к серверной части конструктора.

1.4.4.1 Требования к функциональным характеристикам

Серверная часть конструктора должна предоставлять программный интерфейс, который обеспечивает выполнение следующих функций:

- регистрация пользователей;
- аутентификация пользователей;
- создание ботов;

– вывод списка ботов;
– запуск и остановку ботов;
– добавлять компоненты;
– удалять компоненты;
– редактировать содержимое компонента;
– соединять компоненты;
– обслуживать запросы пользователей от запущенных ботов.

1.4.4.2 Требования к предоставляемому программному интерфейсу

Программный интерфейс, который предоставляется серверной частью конструктора должен удовлетворять следующим требованиям:

– должен быть прост и интуитивен;
– должен быть надежен и доступен;
– должен включать возможность аутентификации и авторизации;
– должен обрабатывать возможные ошибки при запросе пользователя.

1.4.4.3 Требования к параметрам технических и программных средств

В состав технических средств должен входить компьютер, включающий в тебя:

– 64-разрядный процессор с тактовой частотой не менее 1.0 ГГц;
– не менее 4 гигабайт оперативной памяти;
– не менее 1 гигабайт свободного дискового пространства;
– сетевую карту.

Также для работы севера требуется предустановленная операционная система на базе ядра Linux: Ubuntu 18.04 или старше, Debian 10 или старше.

1.4.5 Требования к клиентской части конструктора

В подпунктах данного подраздела описываются требования к клиентской части конструктора.

1.4.5.1 Требования к функциональным характеристикам

Клиентская часть конструктора должна иметь возможность формировать и отправлять данные и запросы для выполнения следующих функций:

– регистрация пользователей;
– аутентификация пользователей;
– создание ботов;

– вывод списка ботов;
– запуск и остановку ботов;
– редактирование ботов.

Для работы с содержимым ботов клиентская часть должна содержать визуальный редактор. Редактор предоставляет пользовательский интерфейс для выполнения следующих функций:

– добавление компонентов;
– удаление компонентов;
– редактирование содержимого компонентов;
– соединение компонентов.

1.4.5.2 Требования к пользовательскому интерфейсу

Интерфейс конструктора Telegram ботов должен состоять из страниц, содержащих разные визуальные элементы, которые предоставляют пользователям возможность взаимодействовать с конструктором.

Также интерфейс должен обеспечивать наглядное, интуитивно понятное представление.

1.4.5.3 Требования к клиентскому программному обеспечению

Клиентская часть конструктора Telegram-ботов должна быть доступна для полнофункционального использования с помощью следующих браузеров:

– Edge 88.0 и выше;
– Opera 43.0 и выше;
– Mozilla Firefox 55.0;
– Google Chrome 64.0 и выше.

1.4.6 Стадии разработки

Разработка должна быть проведена в следующих стадиях:

– разработка технического задания;
– проектирование структуры серверной части конструктора;
– проектирование структуры клиентской части конструктора;
– программная реализация серверной части конструктора;
– программная реализация клиентской части конструктора.

Выводы

В данном разделе был проведен анализ предметной области и осуществлен обзор аналогов. Было выявлено, что многие решения имеют ряд недостатков, таких как ограничения на использование компонентов и показ рекламы. Таким образом, данная тематика и разработка конструктора Telegram-ботов является актуальной.

Из рассмотренных аналогов были выявлены требуемые функциональные возможности разрабатываемого продукта. Было составлено расширенное техническое задание. Определены основные требования к разрабатываемому конструктору Telegram-ботов, такие как требования к структуре, требования к функциональным характеристикам, требования к интерфейсу и к клиентскому аппаратному и программному обеспечению. Также выделены основные стадии разработки.

2 Разработка структуры приложения

На данном этапе работы необходимо в соответствие с требованиями, поставленными в техническом задании, разработать структуру конструктора и описать основные алгоритмы функционирования.

2.1 Разработка общей структуры конструктора

Общая структура конструктора представлена на рисунке 1.

Конструктор состоит из серверной и клиентской части.

Серверная часть состоит из нескольких сервисов:

– сервис пользователей;
– сервис ботов;
– сервис, обслуживающий ботов.

Сервис пользователей взаимодействует с хранилищами в виде базы данных и кэша, в которых хранятся пользователи конструктора и их токены авторизации. Через него происходит регистрация и аутентификация пользователей.

Сервис ботов предоставляет программный интерфейс для работы с ботами. Данный сервис выполняет следующие функции:

– создание ботов;
– редактирование ботов;
– удаление ботов;
– запуск и остановка ботов.

Данные ботов сохраняются в базе данных. Авторизация пользователя происходит с помощью данных из кэша пользователей.

Сервис, обслуживающий ботов, обрабатывает запросы от запущенных Telegram ботов, также он получает и выполняет команды на запуск и остановку ботов от сервиса ботов, команды при этом посылаются через сервер обмена сообщениями. Данный сервис сохраняет и получает текущее состояние пользователя из кэша.

Пользователь взаимодействует с конструктором через веб-интерфейс, ко-

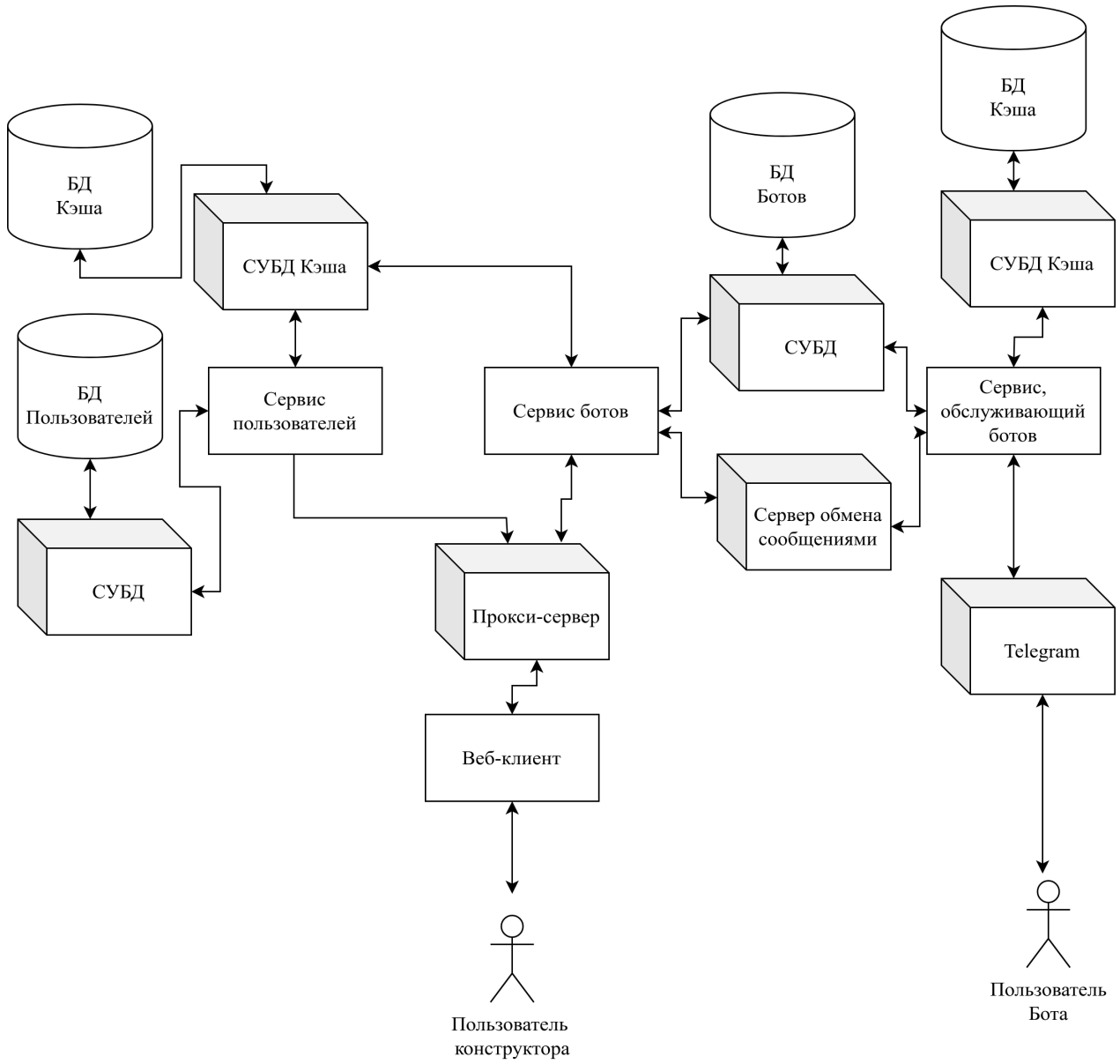

Рисунок 1 – Общая структура конструктора

торый через прокси-сервер связан с сервисами пользователей и ботов. Веб-интерфейс является клиентской частью конструктора.

2.2 Разработка структуры серверной части конструктора

В данном подразделе описываются разработанные структуры серверной части конструктора и его основные алгоритмы функционирования.

2.2.1 Модульная структура серверной части конструктора

Модуль – набор структур и методов, который обобщает какую-то логику приложения.

Модульная структура серверной части конструктора представлена на рисунке 2.

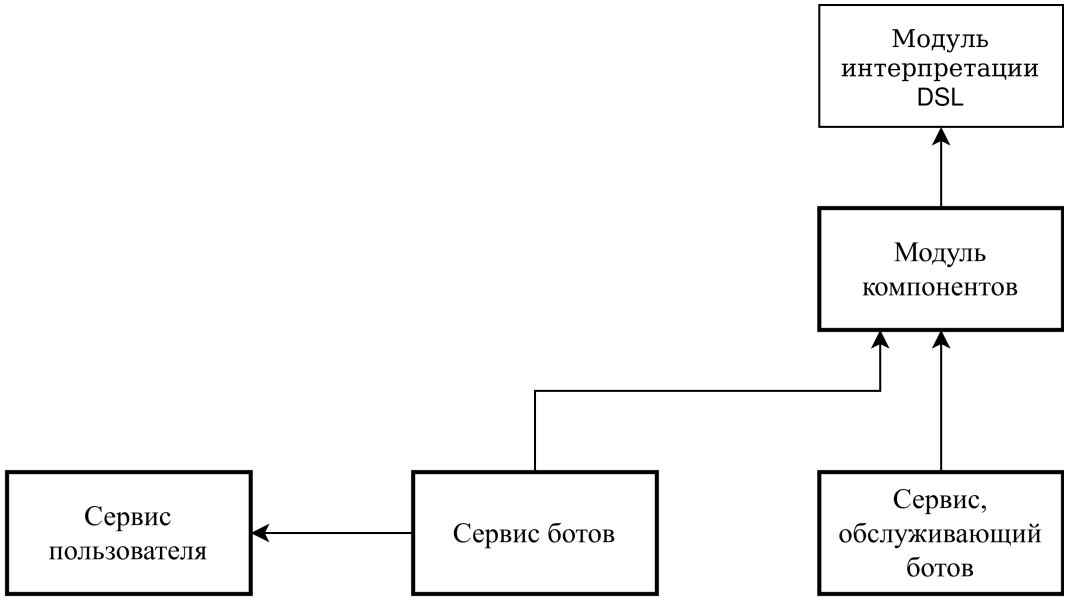

Рисунок 2 – Модульная структура серверной части конструктора

Сервис ботов зависит от сервиса пользователей, который предоставляет первому методы для авторизации пользователя. Также сервис ботов зависит от модуля компонентов, который описывает структуры компонентов и реализует их логику выполнения.

Для выполнения логики ботов сервис, обслуживающий ботов, вызывает методы из модуля компонентов.

2.2.2 Структура модуля компонентов

Модуль компонентов состоит из следующих подмодулей:

– подмодуль компонентов;
– подмодуль контекста;
– подмодуль исполнителя;
– подмодуль ввода-вывода.

Структура модуля компонентов представлена на рисунке 3.

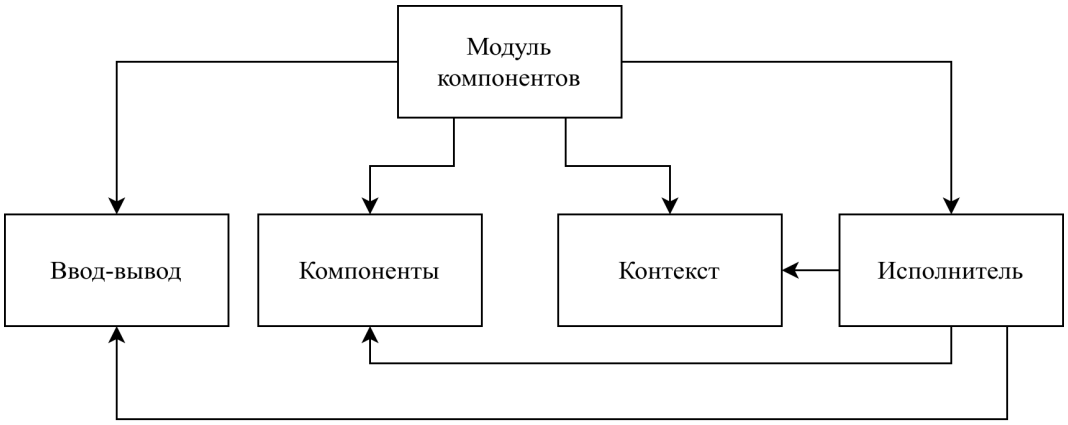

Рисунок 3 – Структура модуля компонентов

Подмодуль компонентов содержит структуру и реализацию логики компонентов. Каждый компонент реализует общий интерфейс компонента, который определен в данном подмодуле.

В данном подмодуле определены следующие компоненты:

– компонент ввода текста;
– компонент отправки сообщения;
– компонент вывода кнопок;
– компонент форматирования;
– компонент точки входа;
– компонент условия.

Подмодуль ввода-вывода предоставляет интерфейс, через который окружение обменивается данными с рядом компонентов.

Подмодуль контекста содержит методы для работы с контекстом. Контекст в рамках бота – память, с которой работают компоненты: компоненты получают из контекста данные для выполнения и записывают в него результат.

Исполнитель представляет собой объект, который содержит в себе контекст и интерфейс ввода-вывода. Через него происходит выполнение компонентов.

2.2.3 Алгоритмы функционирования серверной части конструктора

Пользователь конструктора взаимодействует с сервисом ботом, который контролирует изменение данных и состояние ботов. Схема алгоритма обработки запросов от пользователей сервиса ботов представлен на рисунке 4.

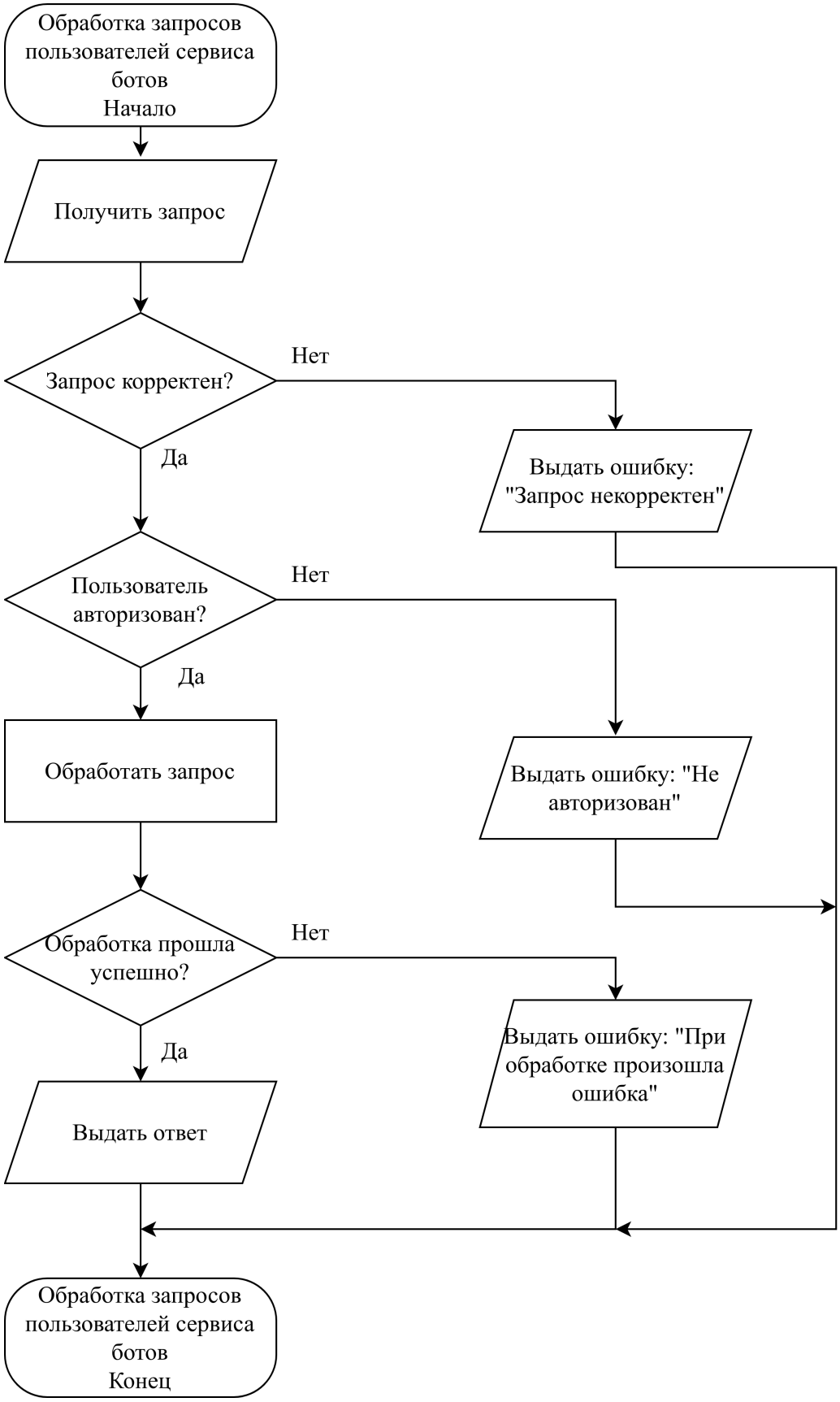

Рисунок 4 – Схема алгоритма обработки запросов от пользователей сервиса ботов

При получении запроса проверяется его корректность. Если запрос не корректен, например, такое обращение не доступно, то выводится соответствующая ошибка. Чтобы обработка прошла успешно, пользователь должен быть авторизован в системе. В случае успешной обработки запроса выдается ответ, иначе - ошибка.

При взаимодействии Telegram пользователя с ботом происходит отправка запросов сервису, обслуживающему ботов, который в дальнейшем обрабатывает данное событие. Схема алгоритма обработки запросов от пользователя бота представлен на рисунке 5.

При принятии события сервисом происходит считывание следующих данных:

– информация о пользователе;
– информация о чате;
– id бота, от которого пришло сообщение.

На основе этих данных из хранилища идёт получение следующих данных:

– контекст пользователя;
– id текущего компонента пользователя;
– компонентов бота.

На основании id текущего компонента происходит получение текущего компонента, который затем выполняется. Результат выполнения представляет собой id следующего компонента, который присваивается текущему.

На основании следующего компонента принимается решение: если компонент ожидает ввода каких-либо данных, то алгоритм заканчивается с сохранением контекста и id текущего компонента, иначе идет выполнение следующего компонента.

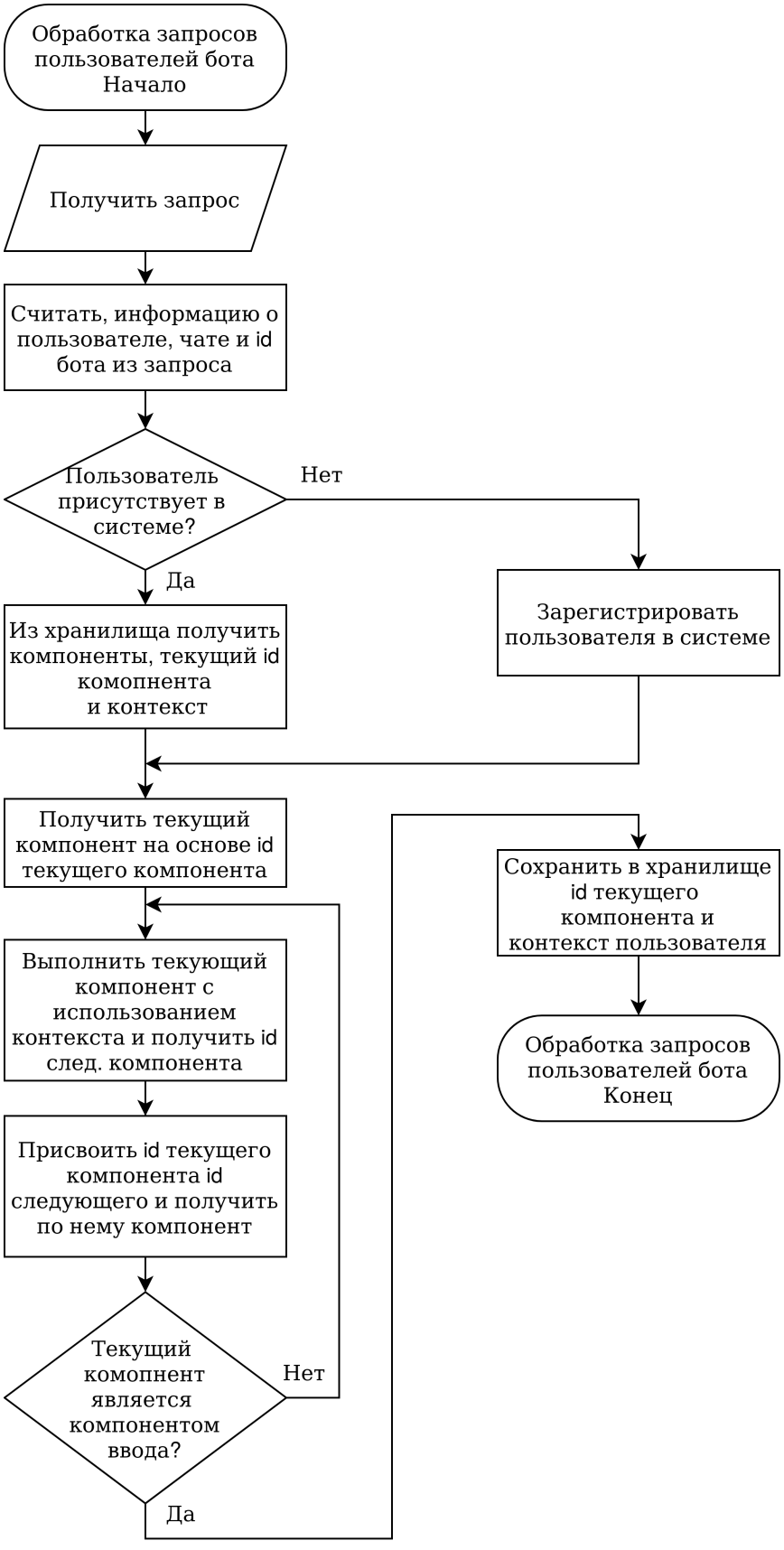

Рисунок 5 – Схема алгоритма обработки запросов от пользователей ботов

2.2.4 Обеспечение защиты информации клиентов конструктора

Для обеспечения защищенного хранения паролей в базе данных используется адаптивная криптографическая хеш-функция bcrypt [3].

Функция основана на шифре Blowfish. Для защиты от атак с помощью радужных таблиц bcrypt использует соль (salt); кроме того, функция является адаптивной, время её работы легко настраивается и её можно замедлить, чтобы усложнить атаку перебором [3].

Функция принимает три параметра: стоимость(cost), соль и пароль. Алгоритм хеширования состоит из следующих шагов [3]:

1) инициализация состояния Blowfish с помощью дорогостоящего алгоритма настройки ключей; результат состоит из массива P, состоящего из 18 подключей, и четырех S-боксов;
2) шифрование текста «OrpheanBeholderScryDoubt» 64 раза с помощью стандартного Blowfish в режиме простой замены;
3) результат состоит из конкатенации стоимости, соли и зашифрованного текста «OrpheanBeholderScryDoubt».

Дорогостоящий алгоритм настройки ключей включает в себя следующие шаги [3]:

1) инициализация подключей P и S-боксов шестнадцатеричными цифрами числа π;
2) перестановка P и S на основе пароля и соли: ExpandKey(P, S, password, salt);
3) повторение 2^{cost} раз следующих шагов:
   3.1) перестановка P и S на основе пароля: ExpandKey(P, S, password, 0);
   3.2) перестановка P и S на основе соли: ExpandKey(P, S, salt, 0).

Алгоритм ExpendKey [3]:

1) смешивание пароля с массивом подключей P;
2) разбиение соли на две равные части;
3) инициализация буфера для хранения блоков;
4) циклическое смешивание внутреннего состояния с P, используя половины соли;

5) циклическое смешивание зашифрованного состояния с внутренними S-блоками состояния, используя половины соли.

Аутентификация пользователя происходит по паролю и логину. Отправленный пароль после хеширования сравнивается с хешем из базы данных, и в случае успеха генерируется токен доступа. Этот токен временно сохраняется в базе данных, а его копия выдается пользователю. Используя токен, пользователь может получать доступ к функциям сервиса ботов.

2.3 Разработка структуры клиентской части конструктора

В данном подразделе описываются разработанные структуры клиентской части конструктора и его диаграммы состояний.

2.3.1 Структура интерфейса конструктора

Пользовательский интерфейс конструктора представляет собой административную панель с набором следующих страниц:

- страница аутентификации;
- страница регистрации;
- страница ботов пользователя конструктора;
- страница создания нового бота;
- страница запуска бота;
- страница редактирования бота.

Каждая страница состоит из верхней панели и содержимого страницы. Верхняя панель содержит ссылки на страницы аутентификации и регистрации, если пользователь не вошёл в систему, иначе - кнопку “выйти из системы”. Шаблон представлен на рисунке 6.

Страница аутентификации и регистрации содержат поля ввода логина и пароля пользователя, под которым располагается кнопка входа или регистрации. Шаблон содержимого страницы аутентификации представлен на рисунке 7.

Страница ботов содержит список блочных элементов, которые включают в себя:

- название бота;

Рисунок 6 – Общий шаблон страниц

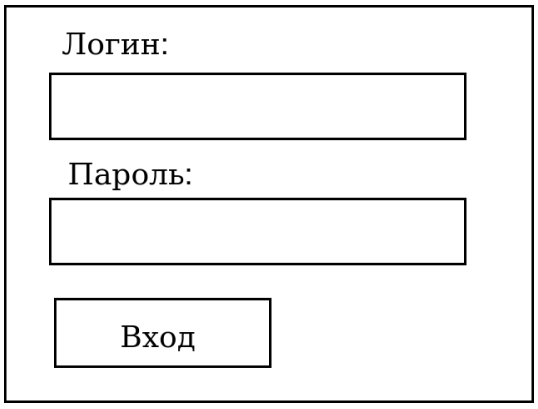

Рисунок 7 – Шаблон содержимого страницы аутентификации

– статус бота;
– кнопка для перехода к редактированию бота;
– кнопка запуска или остановки бота.

Шаблон содержимого страницы списка ботов представлен на рисунке 8.

Страница создания бота содержит одно поле ввода, под которым располагается кнопка создания. Шаблон содержимого представлен на рисунке 9.

Страница запуска бота включает в себя поле ввода токена и кнопку запуска. Шаблон имеет такую же структуру, как и у содержимого страницы создания бота, только с другим именованием кнопки и заголовка поля ввода.

Страница редактирования бота содержит визуальный редактор.

Рисунок 8 – Шаблон содержимого страницы списка ботов

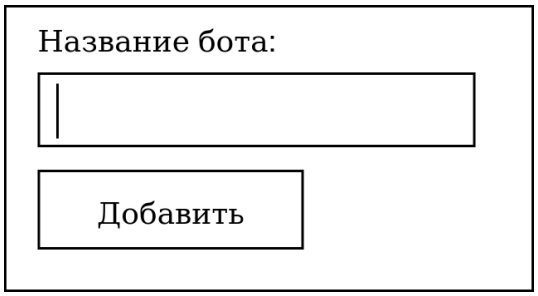

Рисунок 9 – Шаблон содержимого страницы создания бота

2.3.2 Разработка структуры визуального редактора

Визуальный редактор ботов представляет собой область, на которой пользователь может добавлять, редактировать и удалять компоненты, а также связывать их между собой.

2.3.2.1 Модульная структура редактора

Редактор состоит из следующих модулей (Рисунок 10):

– модуль Api-клиента;
– модуль контроллера;
– модуль представления;
– модуль хранилища.

API клиент содержит в себе функции обращения к серверу конструктора для получения и обновления данных бота.

Хранилище содержит методы для изменения данных редактора. При изменении данных хранилища происходит обновление их и на сервере через

API клиент. Контроллер служит посредником между представлением и хранилищем: он содержит обработчики, которые меняют состояние редактора. Ис-

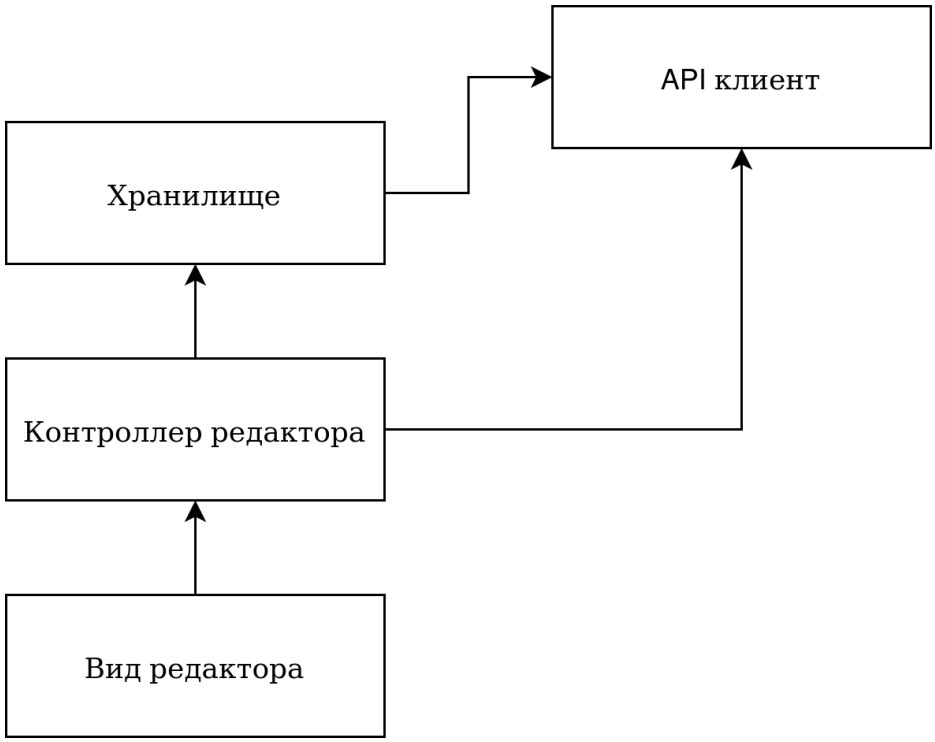

Рисунок 10 – Модульная структура редактора

пользует функции API клиента.

Вид редактора (или представление) содержит в себе компоненты редактора, от которых идут запросы от пользователя. Запросы передаются контроллеру, который их обрабатывает.

2.3.2.2 Компонентная структура редактора

Редактор можно разбить на иерархический набор компонентов: где вышестоящий компонент является родителем, а компонент, который в нем содержится, - дочерним.

Компоненты общаются друг с другом посредством передачи параметров и вызова событий. Родительский компонент вызывает дочерний с помощью передачи параметров, а также отлавливает события дочернего элемента при изменении его состояния.

Компонентная структура визуального редактора представлена на рисунке 11.

На самой высокой ступени стоит компонент редактор. Он хранит все состо-

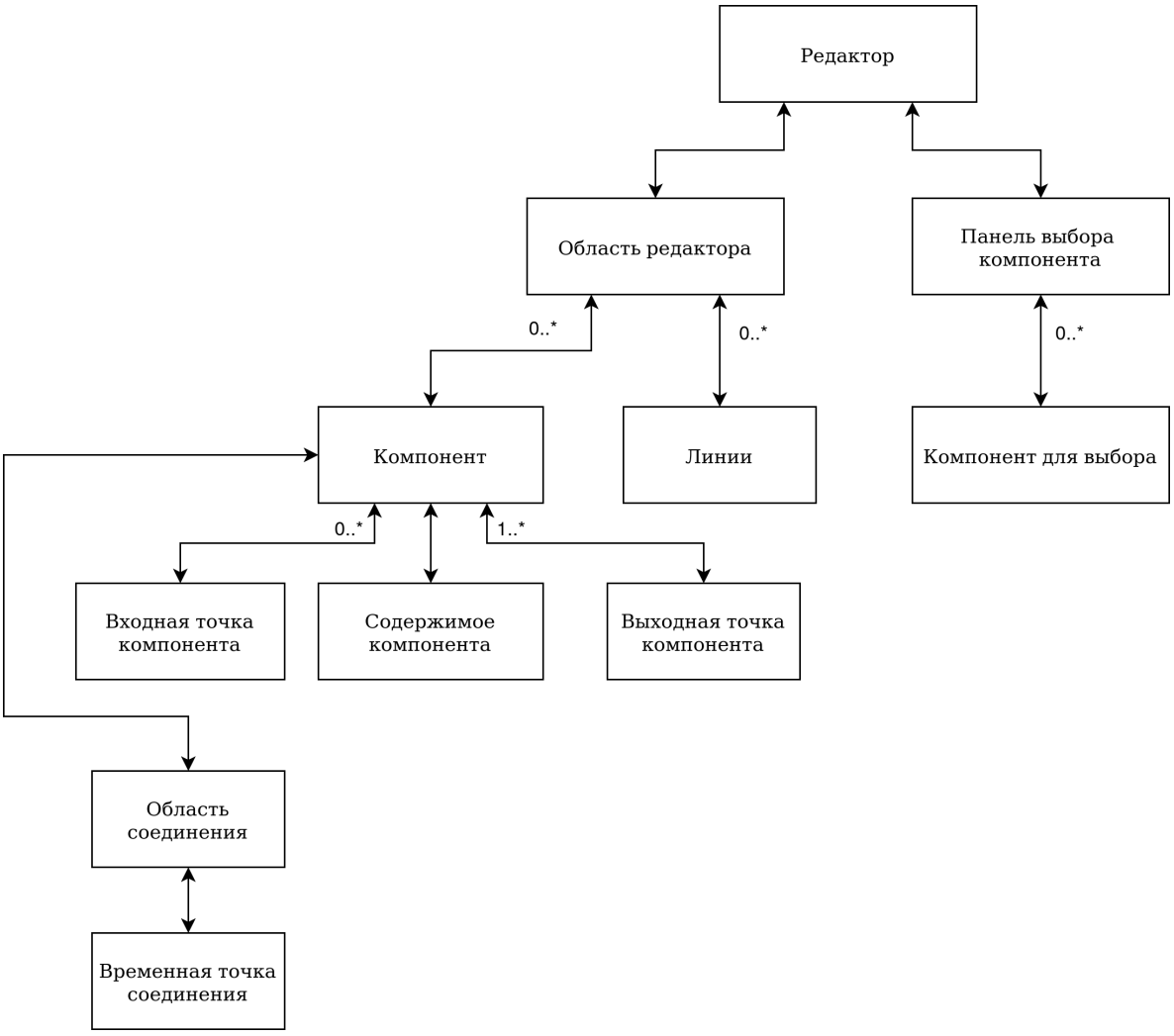

Рисунок 11 – Компонентная структура редактора

яние приложения, а также изменяет его с помощью контроллера. Он состоит из области редактора и панели добавления компонентов.

Панель компонентов содержит разные виды компонентов, которые можно добавить на область редактора путем перетаскивания.

В области редактора содержится набор компонентов бота и их соединений.

Компонент бота содержит в себе следующие компоненты:

– содержимое компонента;
– входные точки компонента;
– выходные точки компонента;
– область соединения.

Компоненты включают в себя разные виды содержимого. Содержимое зависит от типа компонента.

Чтобы контролировать переход по компонентам присутствуют элементы соединений – точки соединения. Благодаря им можно располагать линии между компонентами и тем самым связывать их.

Существует три вида точек соединений:

– входная точка;
– выходная точка;
– временная точка.

Входная точка. Служит для обозначения места соединения у следующего компонента. Может быть несколько – зависит от количества предыдущих компонентов. При нажатии на данный элемент будет происходить событие отвязки.

Выходная точка. Служит для указания следующего компонента. Может быть несколько - зависит от типа компонента. При нажатии на точку будет происходить событие начала соединения.

Временная точка соединения – элемент, который помогает пользователю обозначить место соединения у следующего компонента. Данная точка располагается на области соединения компонента. Вызывает событие конца соединения при отжатии левой кнопки мыши на этом элементе. При этом событии происходит скрытие временной и вставка входной точки.

Область соединения компонента представляет собой место, где возможно расположение входных точек. Область охватывает края компонента.

Также у каждого компонента присутствует кнопка удаления.

Расположение компонентов на шаблоне визуального редактора представлено на рисунке 12.

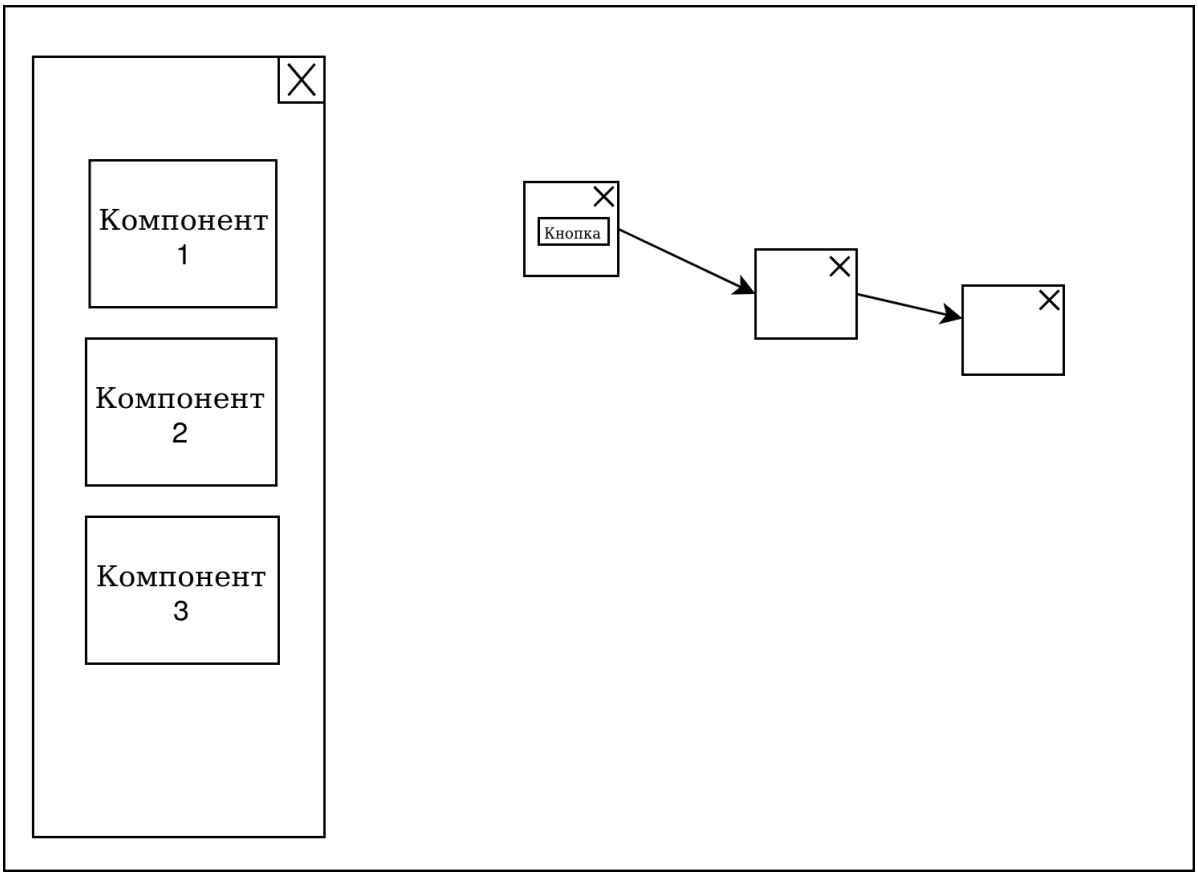

Рисунок 12 – Шаблон визуального редактора

2.3.3 Разработка диаграмм состояний клиентской части конструктора

Диаграммы состояний помогают визуализировать различные состояния системы и переходы между ними, что облегчает понимание её работы. Далее приводятся разработанные диаграммы состояний клиентской части конструктора, а также её основной части - визуального редактора.

2.3.3.1 Диаграмма состояний клиентской части

Клиентская часть конструктора имеет определенный набор состояний. Эти состояния представляют собой страницы, на которых может находиться пользователь. Диаграмма состояний клиентской части конструктора представлена на рисунке 13.

При запуске приложения пользователя ожидает страница входа в систему. Если пользователь не зарегистрирован, то ему предлагается перейти на страницу регистрации.

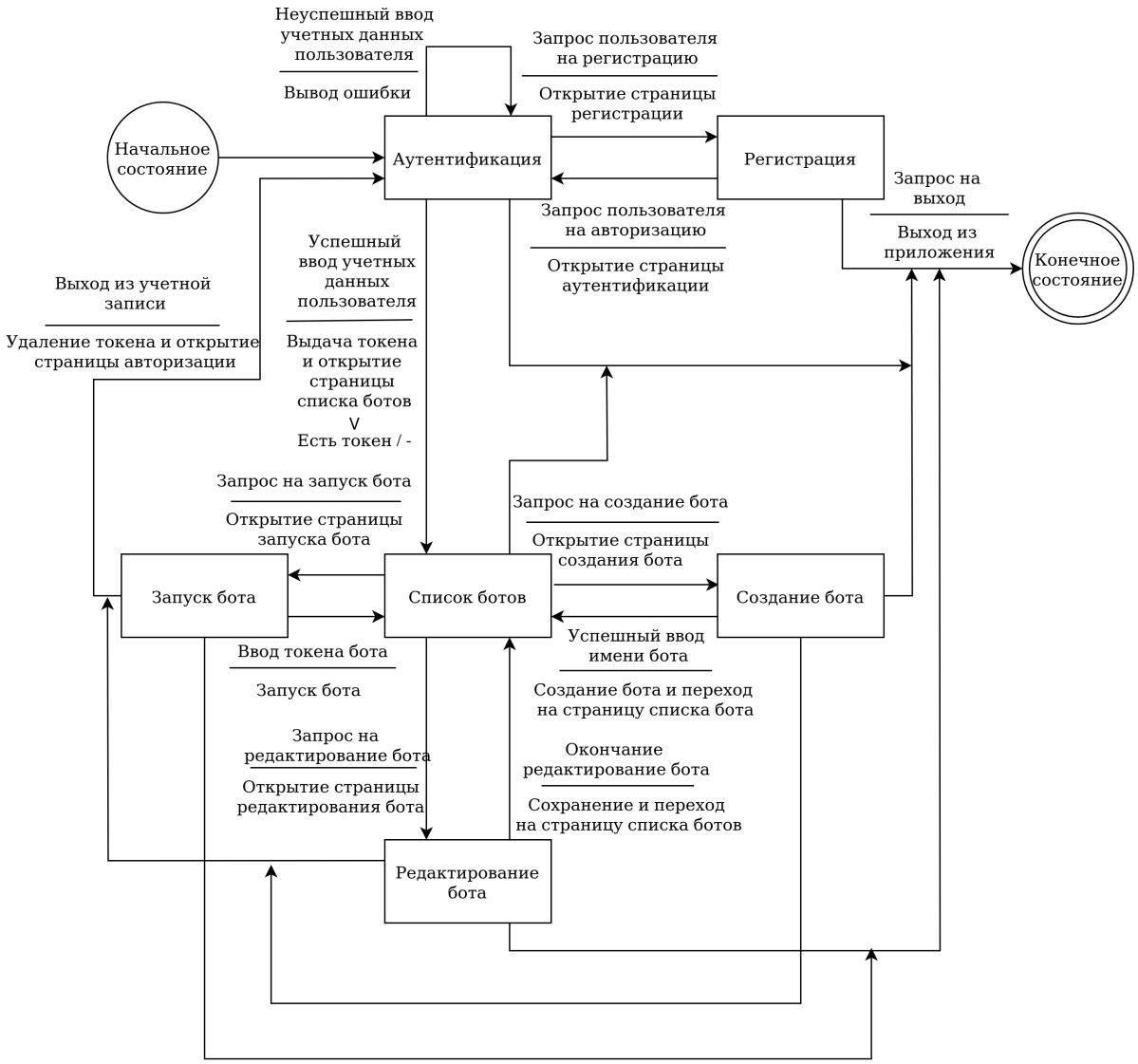

Рисунок 13 – Диаграмма состояний клиентской части конструктора

При правильном вводе логина и пароля пользователю предоставляется токен доступа, и открывается страница списка ботов, иначе – выводится ошибка.

На странице списка ботов пользователю предоставляется выбрать бота для редактирования, удаления или запуска/остановки. Также пользователь может перейти на страницу создания бота.

На странице создания пользователь может ввести название бота, и после нажатия на кнопку “создать” происходит создание бота с последующим возвратом на страницу списка ботов. Список ботов обновляется – в списке уже присутствует вновь созданный бот.

При запуске бота отрывается страница с вводом токена бота. Если токен был

введен успешно, то происходит запуск бота с возвратом на страницу списка ботов.

При нажатии на элемент списка ботов происходит открытие редактора, где пользователь может модифицировать бота: создавать и изменять компоненты, соединять их.

При закрытии любой страницы происходит выход из приложения, также при попытке получения доступа к любой из страниц без валидного токена происходит переход на страницу входа в приложение.

2.3.3.2 Диаграмма состояний визуального редактора

Визуальный редактор также имеет конечный набор состояний. Диаграмма состояний визуального редактора представлена на рисунке 14.

В начале своего запуска визуальный редактор ожидает действия от пользователя, в зависимости от которых происходит изменение его состояния.

В редакторе присутствует область выбора компонента. Если пользователь решит выбрать компонент – инициируется переход в состояние добавления компонента.

В момент перехода в состояние добавления происходит создание временного компонента, после чего пользователь может его перемещать. Если конечное расположение компонента выбрано – инициируется переход обратно в состояние ожидания, при этом временный компонент заменяется постоянным.

Нажимая на любую выходную точку компонента, пользователь дает запрос редактору на переход в режим соединения компонентов, следствием чего является появление линии, которая следует за курсором мыши. Также при этом у всех других компонентов, кроме начального появляется область соединения при наведении мыши на неё. При отпускании левой кнопки мыши на этой области происходит закрепление линии – признак соединения компонентов. Если же отжатие кнопки мыши происходит вне этой области, то линия теряется и соединения не происходит. В любом из этих случаев происходит переход в состояние ожидания.

Отсоединение компонентов происходит при нажатии левой кнопки мыши на входную точку одного из компонентов, и при этом редактор переходит в состояние перемещении линии – состояния соединения компонентов.

При выборе компонента появляется возможность его перемещения. Если пользователь при уже нажатой левой кнопки мыши на компоненте начнет её пе-

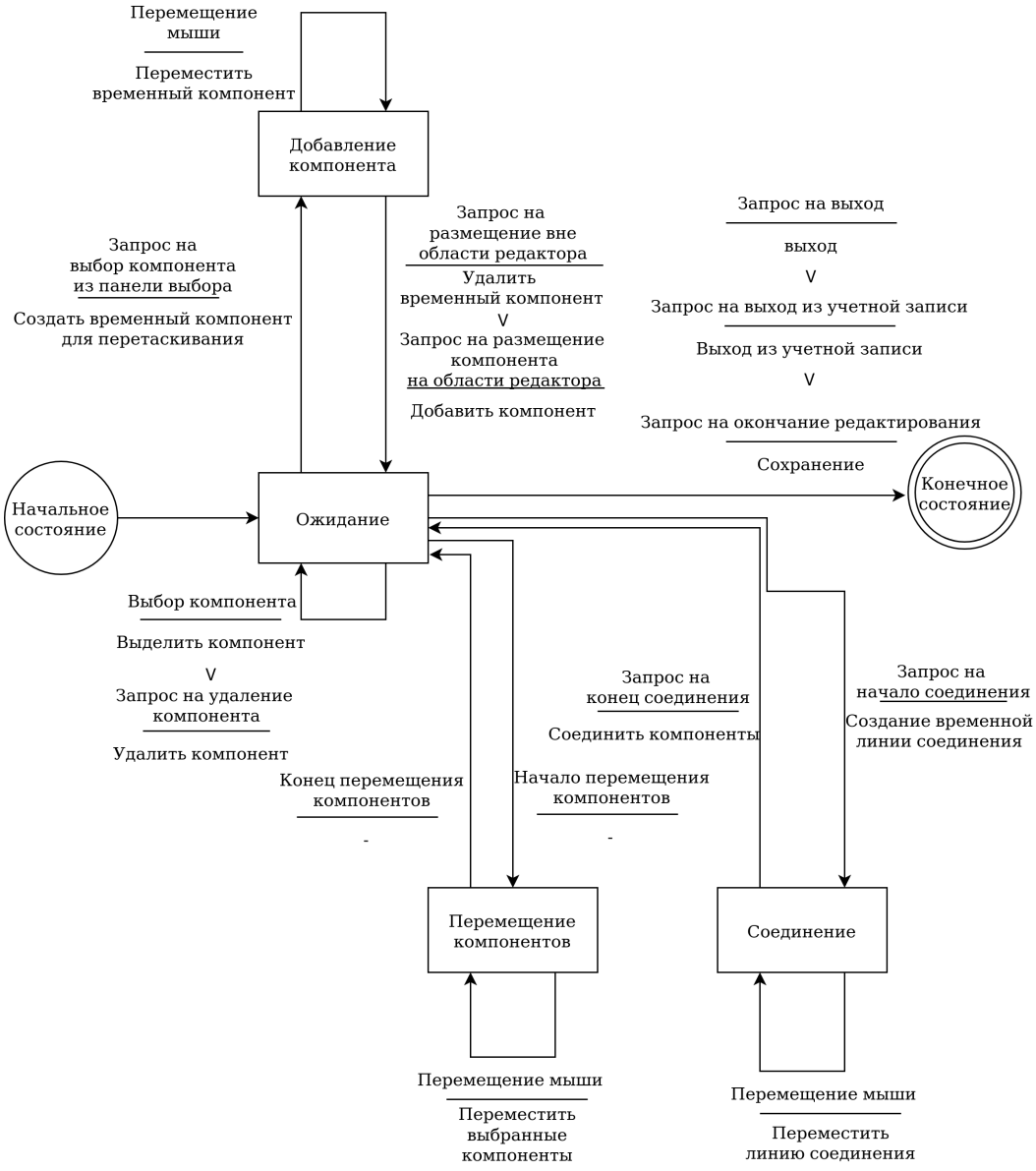

Рисунок 14 – Диаграмма состояний визуального редактора

ремещать, то компонент начнёт перемещение за ней – состояние перемещения. Отпускание левой кнопки мыши закрепит компонент, и редактор снова начнет ожидать действия от пользователя.

2.3.4 Расчёт координат объектов визуального редактора

Область редактора представляет собой координатную плоскость, на которой располагаются компоненты бота. Расположение компонента обеспечивается координатами (x_c, y_c), которые указывают на левый верхний угол компонента.

При перемещении компонента вычисляются смещения $\triangle x$ и $\triangle y$ относи-

тельно координат нажатой мыши (x_m, y_m) по формулам

$$\triangle x = x_c - x_m, \tag{1}$$

$$\triangle y = y_c - y_m. \tag{2}$$

Данные смещения используются для расчёта новых координат компонента (x'_c, y'_c) при перемещении мыши с координатами (x'_m, y'_m), которые вычисляются по формулам

$$x'_c = x'_m - \triangle x, \tag{3}$$

$$y'_c = y'_m - \triangle y. \tag{4}$$

Расположение компонента на координатной плоскости показано на рисунке 15.

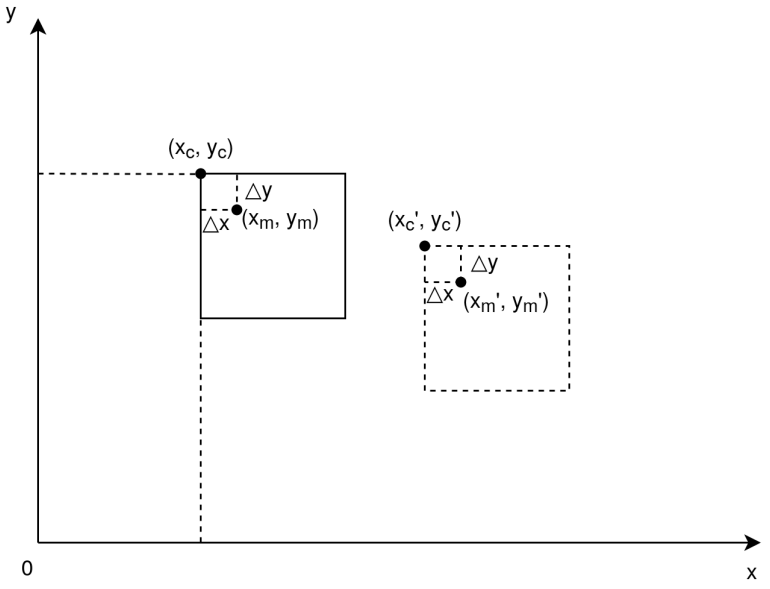

Рисунок 15 – Координаты расположения компонентов

Связи между компонентами представляют собой линию со стрелкой. Линия имеет координаты начала (x_{out}, y_{out}) и конца (x_{in}, y_{in}), которые представляют собой точки центра окружностей соединительных точек выхода и входа компонентов. Расположение связей компонентов на координатной плоскости показано на рисунке 16.

Стрелка представляет собой две примыкающих к линии прямых. Стрелка

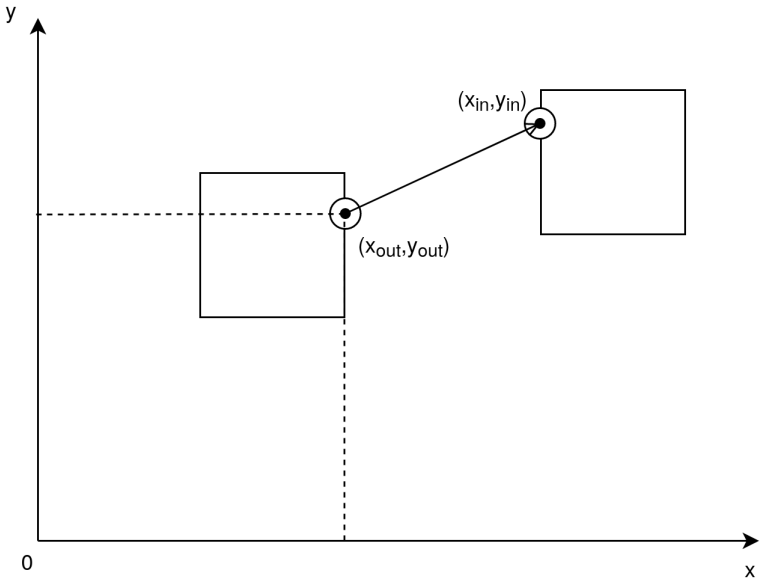

Рисунок 16 – Координаты расположения связей компонентов

имеет длину a и угол между примыкающих прямых α.

Линия между компонентами наклонена под углом β относительно оси y. Угол вычисляется по формуле

$$\beta = \text{arctan}(\frac{x_{in} - x_{out}}{y_{in} - y_{out}}). \tag{5}$$

Точка (x_a, y_a) является окончанием стрелки и её координаты вычисляются по формулам

$$x_a = x_{in} - sin\beta * a, \tag{6}$$

$$y_a = y_{in} - cos\beta * a. \tag{7}$$

Расчет смещения b примыкающих прямых относительно основной линии и смещений $\triangle x_a$ и $\triangle y_a$ относительно осей x и y происходит по формулам

$$b = \text{tan}(\frac{\alpha}{2}) * a, \tag{8}$$

$$\triangle x_a = \text{cos}(\beta) * b, \tag{9}$$

$$\triangle y_a = \text{sin}\,\beta * b. \tag{10}$$

Сами точки окончания примыкающих прямых (x_{a1}, y_{a1}) и (x_{a2}, y_{a2}) вычис-

ляются по формулам

$$x_{a1} = x_a + \triangle x_a, \tag{11}$$

$$y_{a1} = y_a - \triangle y_a, \tag{12}$$

$$x_{a2} = x_a - \triangle x_a, \tag{13}$$

$$y_{a2} = y_a + \triangle y_a. \tag{14}$$

Расположение стрелки на координатной плоскости представлено на рисунке 17.

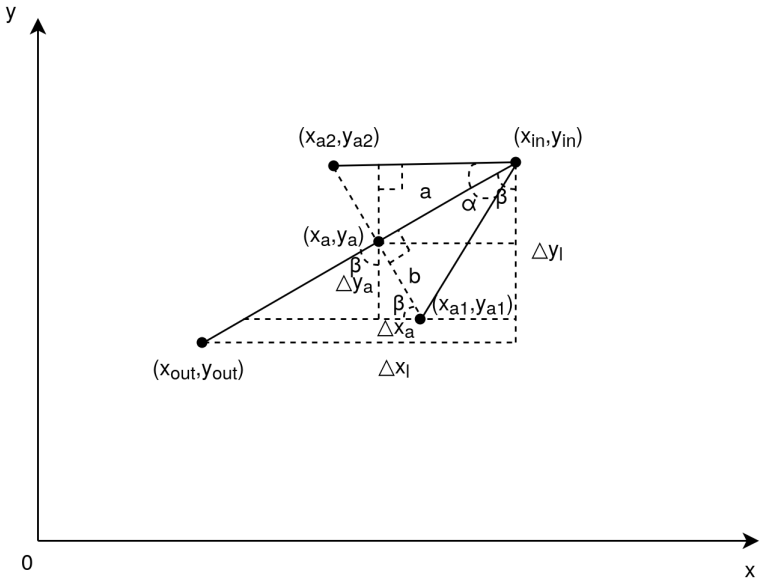

Рисунок 17 – Координаты расположения стрелки

Выводы

В данном разделе была определена визуальная структура серверной и клиентской частей конструктора.

Модульная структура серверной части и визуального редактора позволила выделить определенные функциональные блоки – набор структур и функций, которые ответственны за определенную часть системы. Компонентная структура помогла представить редактор как набор связанных компонентов в виде дерева, эти компоненты взаимодействуют друг с другом.

Были выделены алгоритмы функционирования серверной части конструктора, такие как обработка запросов пользователей сервиса ботов и пользователей ботов. Также было определено поведение клиентской части и входящего в него визуального редактора с помощью диаграммы состояний.

3 Программная реализация

На данном этапе работы необходимо выбрать формат данных, посредством которого будет вестись общение между клиентом и сервером, выбрать инструменты для кодирования приложения (язык программирования, библиотеки), определить интерфейсы для взаимодействия с сервисами и выполнить реализацию серверной и клиентской частей конструктора.

3.1 Формат передаваемых данных

Для взаимодействия клиентской и серверной частей требуется выбрать формат передаваемых данных.

Рассмотрим два формата передачи данных: XML и JSON.

JSON – текстовый формат обмена данными, основанный на JavaScript. Применяется в веб-приложениях как для обмена данными между браузером и сервером, так и между серверами (программные HTTP-сопряжения).

Преимущество данного формата:

– удобен для чтения человеком;

– поскольку JSON является подмножеством синтаксиса языка JavaScript, то он может быть быстро десериализован стандартной библиотекой этого языка на стороне браузера.

XML - формат документа, в котором использованы теги для определения объектов и их атрибутов. Используется для формирования структуры документа и как формат обмена данными.

Преимущества данного формата:

– удобен для создания структуры документа;

– расширяем.

Минусы:

– трудно читаем по сравнению с вышеописанным форматом;

– из-за тегов избыточен при обмене данных.

Из-за удобочитаемости и нативной поддержки в JavaScript для взаимодей-

ствия клиента и сервера был выбран формат JSON.

3.2 Программная реализация серверной части конструктора

Далее идёт обоснование выбора инструментов для реализации серверной части конструктора и описание разработанных программных интерфейсов модулей серверной части, а также её сервисов.

3.2.1 Выбор инструментов разработки

В данном подразделе обосновывается выбор языка программирования для реализации серверной части конструктора, а также системы управления базами данных.

3.2.1.1 Выбор языка программирования

Для разработки сервера в большинстве случаев выбирают следующие языки:

– C++;
– JavaScript;
– PHP;
– Python;
– Golang.

Для выбора языка, на котором будет написан сервер, выделим следующие критерии:

– удобство написания кода;
– быстрота исполнения кода;
– возможность распараллеливания программ;
– доступность библиотек.

C++ - компилируемый, статически типизированный язык программирования. Имеет средства ручного управления памятью, что позволяет достигать высокой скорости работы программы, но при этом усложняется написание кода. Имеет средства для создания потоков, что также может ускорить обработку запросов. Не имеет единой инфраструктуры для управления зависимости приложения.

JavaScript – интерпретируемый, динамически типизированный язык про-

граммирования. Благодаря платформе Node.js способен выполняться на стороне сервера. Однопоточен, распараллеливание достигается путем запуска нескольких экземпляров приложения. Имеет возможность асинхронного выполнения кода. Платформа node.js предоставляет удобный пакетный менеджер – npm. Из-за интерпретации уступает по скорости C++ и другим компилируемым языкам.

PHP – популярный для веб-разработки язык программирования, как и JavaScript является интерпретируемым с динамической типизацией. По скорости также уступает компилируемым языкам, однопоточен. Имеет популярный в кругах языка пакетный менеджер Composer.

Python – довольно популярный и простой в изучении язык программирования, интерпретируемый с динамической типизацией. Имеет пакетный менеджер pip, с помощью которого можно удобно управлять зависимостями приложения. Как и другие рассмотренные интерпретируемые языки программирования однопоточен и уступает по скорости выполнения компилируемым языкам.

Golang – компилируемый язык программирования со статической типизацией. Имеет единую инфраструктуру библиотек и встроенный пакетный менеджер. Несмотря на наличие сборщика мусора, обладает относительно высокой скоростью исполнения. Прост в изучении, имеет простой синтаксис: не перегружен языковыми конструкциями, минималистичен. В языке присутствуют горутины - легковесные потоки.

Сравнение языков приведено в таблице 2.

Таблица 2 – Сравнение языков программирования

	Удобство написания кода	Возможность распараллеливания программы	Быстрота исполнения кода	Доступность библиотек
C++	низкая	есть	высокая	низкая
JavaScript	средняя	нет	средняя	высокая
PHP	средняя	нет	средняя	высокая
Python	высокая	нет	средняя	высокая
Golang	высокая	есть	высокая	высокая

По таблице можно сделать вывод, что самым оптимальным выбором для написания веб-сервера является язык программирования Golang.

3.2.1.2 Выбор системы управления базами данных

Для хранения и управления данными приложения существует множество СУБД, но наиболее популярны такие, как PostgreSQL, MySQL и SQLServer.

SQLServer коммерческая система управления базами данных, которая предоставляет расширенные возможности только при её покупке, что не совсем подходит под условия создания приложения.

PostgreSQL и MySQL – это две популярные реляционные системы управления базами данных с открытым исходным кодом. Ниже приведено сравнение этих двух баз данных по некоторым основным критериям:

– тип данных: обе СУБД поддерживают широкий спектр типов данных, включая текстовые, числовые, даты и времена, бинарные и другие. Однако PostgreSQL имеет более богатый набор типов данных и поддерживает пользовательские типы данных;

– модель данных: обе СУБД используют реляционную модель данных, но PostgreSQL поддерживает более сложные структуры данных, включая хранимые процедуры, триггеры, пользовательские функции и т.д;

– производительность: обе СУБД имеют хорошую производительность, но в некоторых случаях PostgreSQL может быть более производительным, особенно при работе с большими объемами данных;

– масштабируемость: обе СУБД поддерживают горизонтальное и вертикальное масштабирование, но PostgreSQL обычно считается более масштабируемым и подходит для крупных и сложных проектов;

– надежность и безопасность: обе СУБД обеспечивают надежность и безопасность данных, но PostgreSQL обычно обладает более развитыми средствами для обеспечения защиты данных.

Таким образом выбор был сделан в пользу PostgreSQL из-за хорошей производительности, надежности и масштабируемости.

Также для создания серверной части требуется выбрать систему кэширования данных. Наиболее популярны такие СУБД для данных целей, как Memcached и Redis.

Основное различие между Memcached и Redis заключается в их функциональности. Memcached предназначен исключительно для кэширования данных, в то время как Redis обладает более широкими возможностями, такими как возможность работы с различными типами данных, публикация/подписка сообщений, транзакции и другие, поэтому выбор пал именно на него.

3.2.2 Создание базы данных

Для хранения данных приложений была создана база данных представляющая собой следующий набор сущностей:

пользователь конструктора;

- пользователь бота;
- бот;
- группа компонентов;
- компонент.

Пользователь хранит логин и пароль для возможности аутентификации его в системе.

Бот содержит такую информацию, как имя и принадлежность какому- либо пользователю путем указания его идентификатора.

Пользователь бота содержит информацию, которая старается подробно его описать. Сюда входят такие поля, как идентификатор телеграмма, имя, фамилия, ник. Также для хранения текущей сессии пользователя бота содержаться поля идентификатора компонента и контекста.

Группа компонентов содержит поля имени группы, а также поле указания принадлежности к определенному боту. Группы позволяют потенциально расширить возможность конструктора, введя туда компонент, который будет отвечать за определение и вызов функций – набор компонентов.

Компонент содержит поля типа, пути, данных, выходов, идентификатора следующего компонента и позиции на области редактора.

ER-диаграмма представлена на рисунке 18.

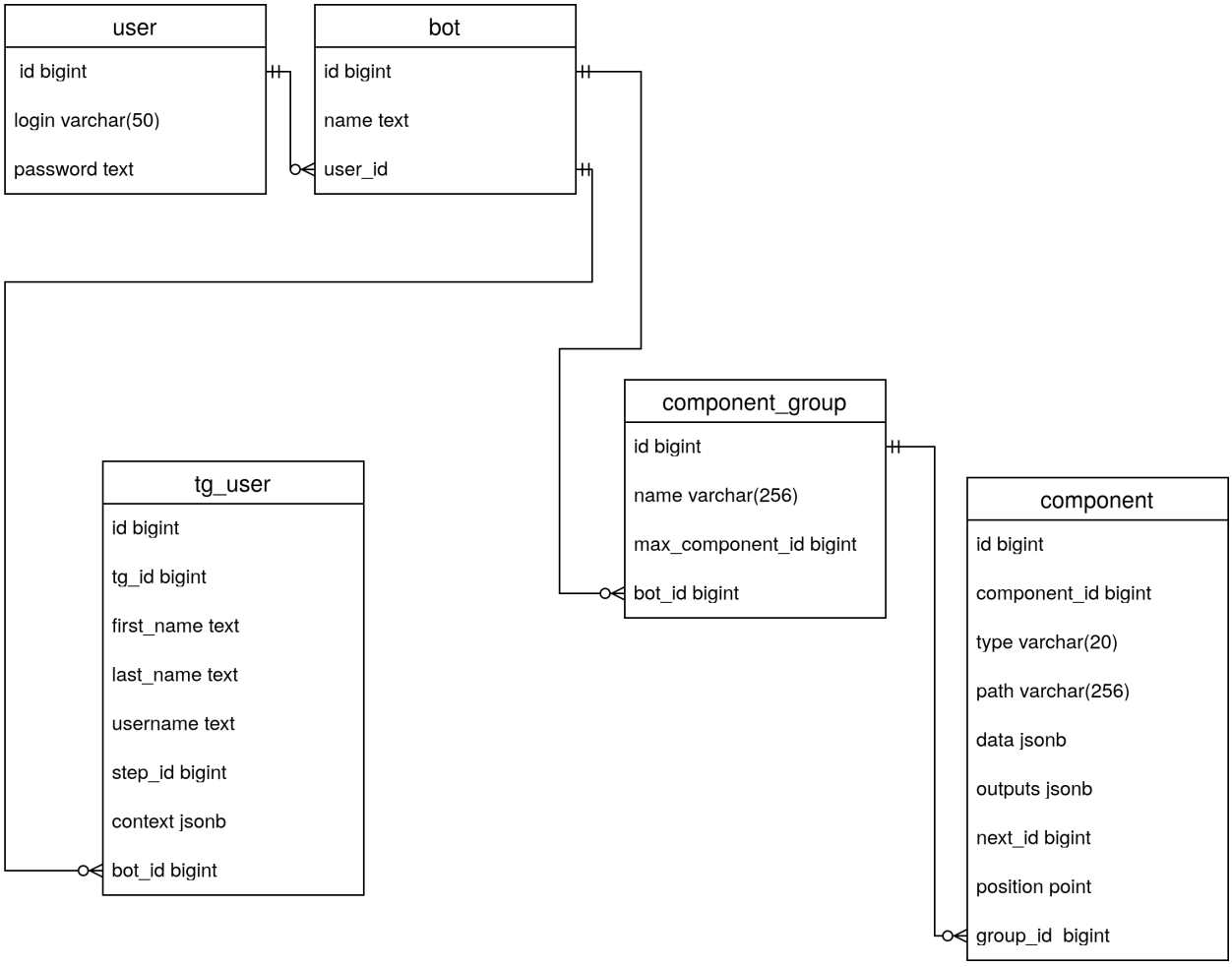

Рисунок 18 – ER-диаграмма

3.2.3 Реализация модуля пользователей

Модуль пользователей предоставляет интерфейс и его реализацию для авторизации пользователей в системе.

Интерфейс представлен на рисунке 19.

Функция сохранения токена принимает токен, время его жизни и сохраняет его в хранилище, который определяется реализацией интерфейса.

Функция удаления удаляет токен из хранилища.

Функция проверки позволяет удостовериться в валидности принимаемого токена.

Функция закрытия служит для закрытия соединения с определенным в реализации хранилищем.

Данный интерфейс используется сервисом ботов и сервисом пользователей.

```
type TokenStorage interface {
        SaveToken(ctx context.Context,
              token string,
              lifeDuration time.Duration) error

        DeleteToken(ctx context.Context,
                token string) error

        CheckToken(ctx context.Context,
               token string) (bool, error)

        Close() error
}
```

Рисунок 19 – Интерфейс, предоставляющий возможность авторизации пользователей

3.2.4 Реализация модуля компонентов

Модуль компонентов содержит реализацию компонентов конструктора, а также предоставляет интерфейсы для работы с ними

3.2.4.1 Интерфейсы модуля

Компоненты имеют следующую общую структуру, представленную на рисунке 20.

Структура содержит следующие поля:

– поле id идентифицирует компонент;

– поле type содержит тип компонента в виде строки;

– поле path предназначен для хранения пути данных для контекста, оно будет использоваться в зависимости от компонента: по нему будут или записываться данные, или считываться для последующего использования;

– поле data содержит данные, которые будут использоваться компонентом. Данные индивидуальны для каждого компонента;

– поле outputs содержит выходы компонента - id следующего возможного компонента. Часть полей общие, а остальные индивидуальны.

Общее поле outputs для всех компонентов - id следующего компонента. Для неко-

```
{
    id: number,
    type: string,
    path: string,
    data: {
        <field1>: any,
        ...
    },
    outputs: {
        <field1>: number,
        <field2>: number,
        ...
    }
}
```

Рисунок 20 – Общая структура компонентов

торых компонентов задаётся только во время выполнения.

Если в поле outputs не определен какой-либо выход, который требуется для какого-либо компонента, ему присваивается пустое значение.

Объект компонента создается функцией, представленной на рисунке 21.

```
func NewComponentFromJSON(
    tp ComponentType,
    jsonData []byte
    ) (Component, error)
```

Рисунок 21 – Функция создания компонента

Функция принимает тип компонента в виде строки и данные json, представленные в виде массива байтов.

Каждый компонент имеет свою логику выполнения. Выполнением компонентов занимается объект Executor (Исполнитель).

Исполнитель состоит из контекста и интерфейса ввода-вывода. Создание объекта исполнителя происходит с помощью функции, представленной на рисун-

ке 22.

```
func NewExecutor(ctx *Context, io IO) *Executor
```

Рисунок 22 – Функция создания исполнителя

Исполнитель имеет лишь один метод, представленный на рисунке 23.

```
func (e *Executor) Execute(
        component Component) (*int64, error)
```

Рисунок 23 – Метод выполнения компонентов

В качестве параметра выступает интерфейс компонента. В данном методе происходит выполнение логики компонента. Компонент выполняется в зависимости от реализации интерфейса определенного вида компонента. Результатом является id следующего компонента.

Передача данных между компонентами происходит через Context. Так как структура данных контекста изначально не определена, внутри Context находятся данные неопределенного типа.

Объект Context создаётся из JSON методом, представленным на рисунке 24.

```
func NewContextFromJSON(jsonData []byte) (*Context, error)
```

Рисунок 24 – Функция создания контекста

Интерфейс ввода-вывода используется при выполнении компонентов. Он позволяет взаимодействовать со средой, в которой выполняются компоненты.

Интерфейс ввода-вывода имеет вид, представленный на рисунке 25.

```
type IO interface {
        PrintText(text string)
        PrintButtons(text string, buttons []*ButtonData)
        ReadText() *string
}
```

Рисунок 25 – Интерфейс ввода-вывода

Окружение, которое будет использовать компоненты, должна реализовать данный интерфейс.

Интерфейс состоит из следующих методов:

– PrintText - Вывод текста пользователю;

– PrintButtons - Вывод кнопок пользователю с указанием текста опроса;

– ReadText - Считывание текста от пользователя, если текст не введен, возвращается нулевое значение - nil.

3.2.4.2 Список компонентов

В модуле были реализованы следующие компоненты:

– стартовый компонент;

– форматирование;

– сообщение;

– ввод текста;

– условие;

– код;

– http;

– фото;

– кнопки.

Стартовый компонент служит точкой запуска бота. Его основная задача - вернуть следующий компонент для выполнения. У компонентов нет данных, выход лишь один – nextComponentId (id следующего компонента).

Компонент форматирования служит для преобразования текста к нужному формату посредством вставки необходимых данных. Имеет одно поле данных – formatString (строку форматирования). В качестве выхода служит поле

nextComponentId.

Для вставки данных из контекста служит конструкция, представленная на рисунке 26.

```
\${<getting data from context>}
```

Рисунок 26 – Конструкция в тексте, которая позволяет вставлять данные из контекста в компоненте форматирования

Компонент сообщение отправляет текст пользователю. Отправка происходит через интерфейс ввода-вывода. Данными служит выводимый текст – поле text, выход одни – nextComponentId.

Компонент ввода текста ожидает ввода текста от пользователя. Получение текста происходит через интерфейс ввода-вывода. У компонента нет данных, выход один – nextComponentId.

Компонент условия проверяет переменную из контекста: если в ней содержится истинность, то идёт переход на следующий компонент, если ложь - переход происходит по другому выходу.

Компонент кода позволяет пользователю расширить функционал компонентов путём написания небольших скриптов.

Компонент http позволяет обращаться к другим сервисам через соответствующий протокол. В качестве параметров выступают адрес, метод, заголовки и тело запроса.

Компонент фото отправляет картинку пользователю. Параметрами являются название и картинка, представляющая собой массив байт.

Компонент кнопок выводит набор кнопок для пользователя. Вывод кнопок происходит через интерфейс ввода-вывода. Содержит данные, представленные на рисунке 27.

Выходы компонента кнопок представлены на рисунке 28. Имена выходов совпадают с элементами поля buttons в data.

```
type Button = {
    text: string;
}

data: {
    text: string,
    buttons: {
        <numeric field name>: Button,
        ...
    }
}
```

Рисунок 27 – Данные компонента кнопок

```
outputs: {
    <numeric field name>: number,
    ...
}
```

Рисунок 28 – Выходы компонента кнопок

3.2.5 Реализация сервисов

Сервисы предоставляют API для обеспечения выполнения основных функций конструктора. Далее приводится описание их программных интерфейсов.

3.2.5.1 Коды ответов

Сервисы работают по протоколу HTTP и имею следующие возможные коды ответов [8]:

– 200 (Ok) - Успешный ответ с содержимим в теле;

– 201 (Created) - Успешный ответ после создания ресурса. Может содержать информацию о вновь созданном ресурсе;

– 204 (No Content) - Успешный ответ без содержимого в теле;

– 400 (Bad Request) - Неверно составлен запрос;

– 401 (Unauthorized) - Не авторизован;

– 404 (Not Found) - Ресурс не найден;

– 422 (Unprocessable Entity) - Ошибки бизнес логики. В теле ответа содержится подробное описание ошибки;

– 500 (Internal Server Error) - Внутренняя ошибка сервера.

Формат ошибки в теле ответа, если код равен 422, представлена на рисунке 29.

```
{
    code: integer,
    message: string
}
```

Рисунок 29 – Формат ошибки в случае кода ответа 422

Формат ошибки содержит два значения – кода ошибки и сообщения, которое описывает ошибку.

3.2.5.2 Программный интерфейс сервиса пользователей

Сервис пользователей предоставляет следующие запросы:

– регистрация:

Запрос: POST /api/users/signup.

Структура тела запроса представлена на рисунке 30.

```
{
    "login": "string"
    "password": "string"
}
```

Рисунок 30 – Тело запроса регистрации

Ответ: в случае успеха - статус кода HTTP - 201. В случае некорректных данных - статус кода HTTP - 422 с описанием ошибки в теле ответа;

– аутентификация:

Запрос: POST /api/users/signin.

Структура тела запроса представлена на рисунке 30.

Ответ: В случае успеха - статус кода HTTP - 201 с телом ответа, который представлена на рисунке 31.

```
{
    "token": "string"
}
```

Рисунок 31 – Тело ответа при аутентификации в случае успеха

В случае некорректных данных - статус кода HTTP - 422 с описание ошибки в теле ответа;

– выход из системы:

Запрос: DELETE /api/users/signout.

Ответ: В случае успеха - статус кода HTTP - 204. Если пользователь не авторизован - 401.

3.2.5.3 Программный интерфейс сервиса ботов

Сервис ботов предоставляет следующий интерфейс:

– создание бота:

Запрос: POST /api/bots.

Тело запроса представлено на рисунке 32.

```
{
    "token": "string"
}
```

Рисунок 32 – Тело запроса при создании бота

Код ответа в случае успеха – 201, тело ответа представлено на рисунке 33;

– получение списка ботов:

Запрос: GET /api/bots.

```
{
    "botId": "integer",
    "component": "component"
}
```

Рисунок 33 – Тело ответа при создании бота

В случае успеха код ответа 200, с телом ответа, содержащим список ботов, структура которых представлена на рисунке 34;

```
{
    "id": "integer",
    "title": "string",
    "status": "integer"
}
```

Рисунок 34 – Структура бота

– удаление бота:

Запрос: DELETE /api/bots/botId.

В случае успеха код ответа 204;

– установка токена бота:

Запрос: POST /api/bots/botId/token.

Тело запроса представлено на рисунке 35.

В случае успеха код ответа 204;

```
{
    "token": "string"
}
```

Рисунок 35 – Тело запроса при установке токена бота

– получение токена бота: Запрос: GET /api/bots/id/token.

В случае успеха код ответа 200 с телом ответа, содержащим токен бота в виде строки;

– запуск бота:

Запрос: PATCH /api/bots/id/start.

В случае успеха код ответа 204;

– остановка бота:

Запрос: PATCH /api/bots/id/stop.

В случае успеха код ответа 204;

– получение компонентов бота:

Запрос: GET /api/bots/botId/groups/groupId/components

В случае успеха код ответа 200 с телом ответа, содержащим список компонентов, структура которых представлена на рисунке 36;

– создание компонента:

Запрос: POST /api/bots/botId/groups/groupId/components.

Тело запроса представлено на рисунке 37.

Тело ответа содержит id вновь созданного компонента в случае успеха с кодом ответа 201;

– удалить компонент:

Запрос: PATCH /api/bots/botId/groups/groupId/components/compId/data.

В случае успеха код ответа 204;

– изменить данные компонента:

Запрос: PATCH /api/bots/botId/groups/groupId/components/compId/data.

Тело запроса представлено на рисунке 38.

В случае успеха код ответа 204;

– изменить позицию компонента:

Запрос: PATCH /api/bots/botId/groups/groupId/components/compId/position.

Тело запроса представлено на рисунке 39.

В случае успеха код ответа 204;

– добавить соединение между компонентами:

Запрос: POST /api/bots/botId/groups/groupId/connections.

Тело запроса представлено на рисунке 40.

```
{
    "id": "integer",
    "type": "string",
    "data": "object",
    "path": "string",
    "outputs": {
        "nextComponentId": "integer",
        ...
    },
    "connectionPoints": {
        "<pointId: string>": {
            "sourceComponentId": "integer",
            "sourcePointName": "string",
            "relativePointPosition": {
                "x": "integer",
                "y": "integer"
            }
        }
        ...
    },
    "position": {
        "x": "integer",
        "y": "integer"
    }
}
```

Рисунок 36 – Структура компонента при ответе на запрос

```
{
    "type": "string",
    "position": {
        "x": "integer",
        "y": "integer"
    }
}
```

Рисунок 37 – Тело запроса при создании компонента

В случае успеха код ответа 204;

– добавить соединение между компонентами:

```
{
    "<property name>": "any",
    ...
}
```

Рисунок 38 – Тело запроса при изменении данных компонента

```
{
    "x": "integer",
    "y": "integer"
}
```

Рисунок 39 – Тело запроса при изменении позиции компонента

```
{
    "sourceComponentId": "integer",
    "sourcePointName": "string",
    "targetComponentId": "integer",
    "relativePointPosition": {
        "x": "integer",
        "y": "integer"
    }
}
```

Рисунок 40 – Тело запроса при добавлении соединения между компонентами

Запрос: DELETE /api/bots/botId/groups/groupId/connections.

Тело запроса представлено на рисунке 41.

В случае успеха код ответа 204.

Если пользователь ввел невалидные данные в любой из приведенных выше запросов, будет код ответа – 422.

При обращении к API каждый запрос должен содержать следующий заго-

```
{
    "sourceComponentId": "integer",
    "sourcePointName": "string"
}
```

Рисунок 41 – Тело запроса при удалении соединения между компонентами

ловок: “Authorization: Bearer <token>”.

3.3 Программная реализация клиентской части конструктора

В данном подразделе обосновывается выбор инструментов разработки клиентской части конструктора, а также демонстрируется итоговый дизайн пользовательского интерфейса.

3.3.1 Выбор инструментов разработки

В браузере доступны следующие инструменты для создания веб-приложения:

– HTML;

– CSS;

– JavaScript.

HTML (HyperText Markup Language — «язык гипертекстовой разметки») — самый базовый строительный блок Веба. Он определяет содержание и структуру веб-контента [9].

Cascading Style Sheets (CSS) — это язык иерархических правил, используемый для представления внешнего вида документа, написанного на HTML. CSS описывает, каким образом элемент должен отображаться на экране, на бумаге, голосом или с использованием других медиа средств [10].

JavaScript — это легковесный, интерпретируемый или JIT- компилируемый, объектно-ориентированный язык с функциями первого класса. Наиболее широкое применение находит как язык сценариев веб- страниц. JavaScript это прототипно-

ориентированный, мультипарадигменный язык с динамической типизацией, который поддерживает объектно- ориентированный, императивный и декларативный (например, функциональное программирование) стили программирования [11].

Основной недостаток JavaScript – это динамическая типизация, при которой переменная связывается с типом в момент присваивания значения, а не в момент объявления переменной. Это особенность JS приводит к достаточно долгой отладке кода при возникновении ошибки.

Для решения этой проблемы был выбран язык TypeScript – компилируемый язык с возможностью явного статического назначения типов. Статическая типизация устраняет основной недостаток JS, а компиляция позволяет выявить некоторые ошибки до запуска приложения. Также он совместим с JS [12].

Среди инструментов, облегчающих разработку клиентских приложений в браузере, имеются:

– Vue - JavaScript-фреймворк с открытым исходным кодом для создания пользовательских интерфейсов. Легко интегрируется в проекты с использованием других JavaScript-библиотек. Может функционировать как веб-фреймворк для разработки одностраничных приложений в реактивном стиле.

– Angular - открытая и свободная платформа для разработки веб- приложений, написанная на языке TypeScript, разрабатываемая командой из компании Google, а также сообществом разработчиков из различных компаний.

– React - JavaScript-библиотека с открытым исходным кодом для разработки пользовательских интерфейсов. React разрабатывается и поддерживается Facebook, Instagram и сообществом отдельных разработчиков и корпораций. React может использоваться для разработки одностраничных и мобильных приложений. Его цель — предоставить высокую скорость разработки, простоту и масштабируемость.

– SolidJS – это легковесная JavaScript библиотека для создания пользовательских интерфейсов. SolidJS вдохновлен библиотекой React, но нацелен на предоставление более простой и эффективной модели программирования.

В качестве сравнения данных библиотек и фреймворков выделим следующие критерии:

– поддержка реактивности;

– сложность изучения;
– производительность;
– потребление памяти.

Под реактивностью понимается обновления отображения при изменении привязанных данных.

Angular является фреймворком для создания крупных браузерных решений. Он имеет довольно много возможностей и из-за этого имеет довольно большой размер и сложен для обучения.

Vue является фреймворком для создания реактивных приложений. Имеет много оптимизаций в своей основе: кэширование вычисляемых свойств, умная перерисовка – перерисовываются только нужные узлы, что позволяет работать приложению довольно быстро. Также легок в обучении и не требует много памяти.

React является библиотекой для создания реактивных приложения. Имеет немного низкую производительность, чем Vue, но обладает высокой скоростью обучения и также предрасположен к низкому потреблению памяти.

SolidJS также является библиотекой для создания реактивных приложений. Реактивность реализуется без использования виртуального DOM, что увеличивает производительность библиотеки и уменьшает потребление памяти.

Сравнение фреймворков и библиотек по критериям представлено в таблице 3.

Таблица 3 – Сравнение фреймворков и библиотек JavaScript

	Vue.js	React	Angular	SolidJS
Поддержка реактивности	есть	есть	есть	есть
Сложность изучения	низкая	низкая	высокая	низкая
Производительность	средняя	средняя	средняя	высокая
Потребление памяти	низкое	низкое	высокое	низкое

Исходя из таблицы можно сделать вывод, что SolidJS является самым оптимальным выбором для разработки клиентской части для конструктора Telegram-ботов.

3.3.2 Дизайн пользовательского интерфейса

На основании составленных шаблонов для интерфейса клиентской части была выполнена их реализация и стилизация.

Для обозначения элементов интерфейса конструктора Telegram-ботов были выбраны следующие цвета:

- серый;
- синий;
- зелёный;
- красный;
- желтый.

Также у каждых цветов были выбраны более светлые оттенки для обозначения фонов неактивных элементов, таких как кнопки.

Преобладающий цвет – синий, он является основным для конструктора. Он, а также его более светлые оттенки используются для обозначения компонентов в редакторе бота, а также для стилизации шапки веб-приложения. Также данный цвет служит для обозначения кнопок, которые обеспечиваю переход на другие страницы.

Серый цвет является основным фоновым для веб-приложения, на нём располагаются белые плитки. Плитка представляет собой группу элементов интерфейса конструктора.

Зелёный цвет применяется для обозначения кнопок, благодаря которым будет происходить добавление объектов конструктора или изменение их состояния на активное.

Красный цвет служит для метки тех элементов, которые способны удалить объекты конструктора, или для обозначения выходов ошибок у компонентов.

Жёлтый цвет применяется для обозначения элементов интерфейса, которые позволяют вернуться назад по страницам, либо служат для изменения состояния объектов конструктора на неактивное или для их индикации.

Экранные формы пользовательского интерфейса конструктора представле-

на на рисунках 42-46.

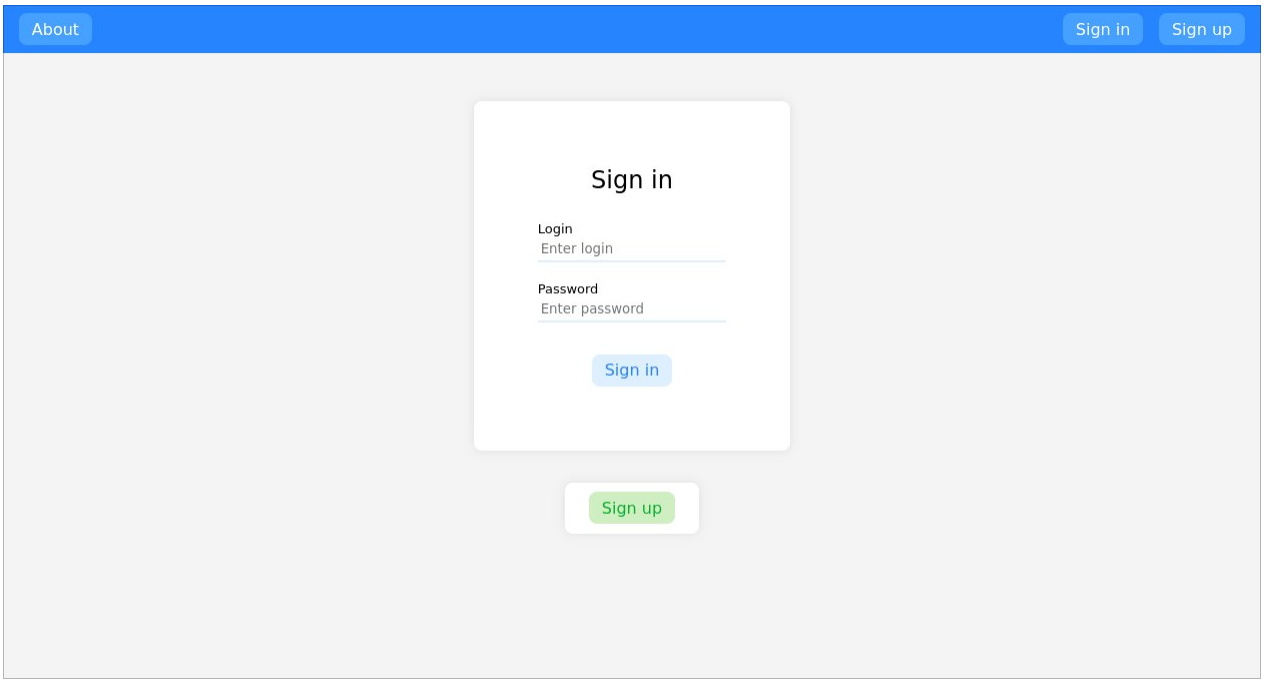

Рисунок 42 – Страница входа в приложение

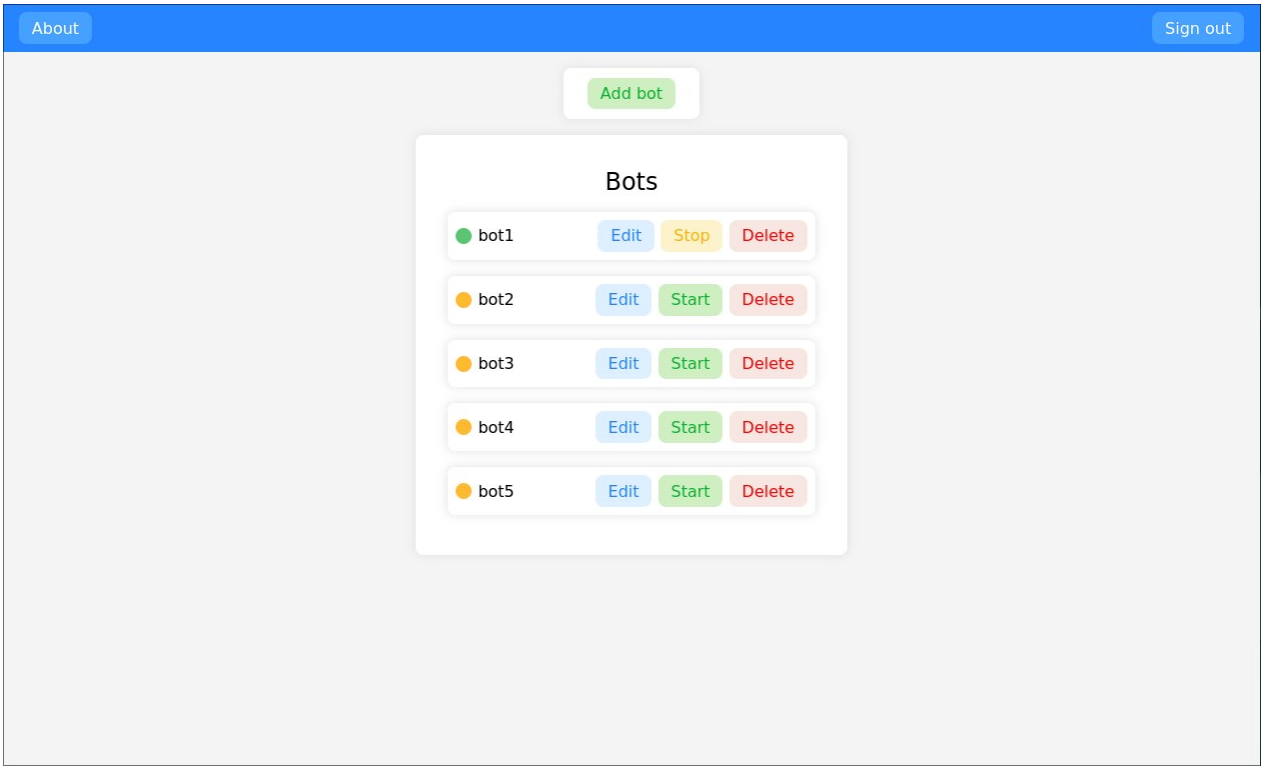

Рисунок 43 – Страница списка ботов

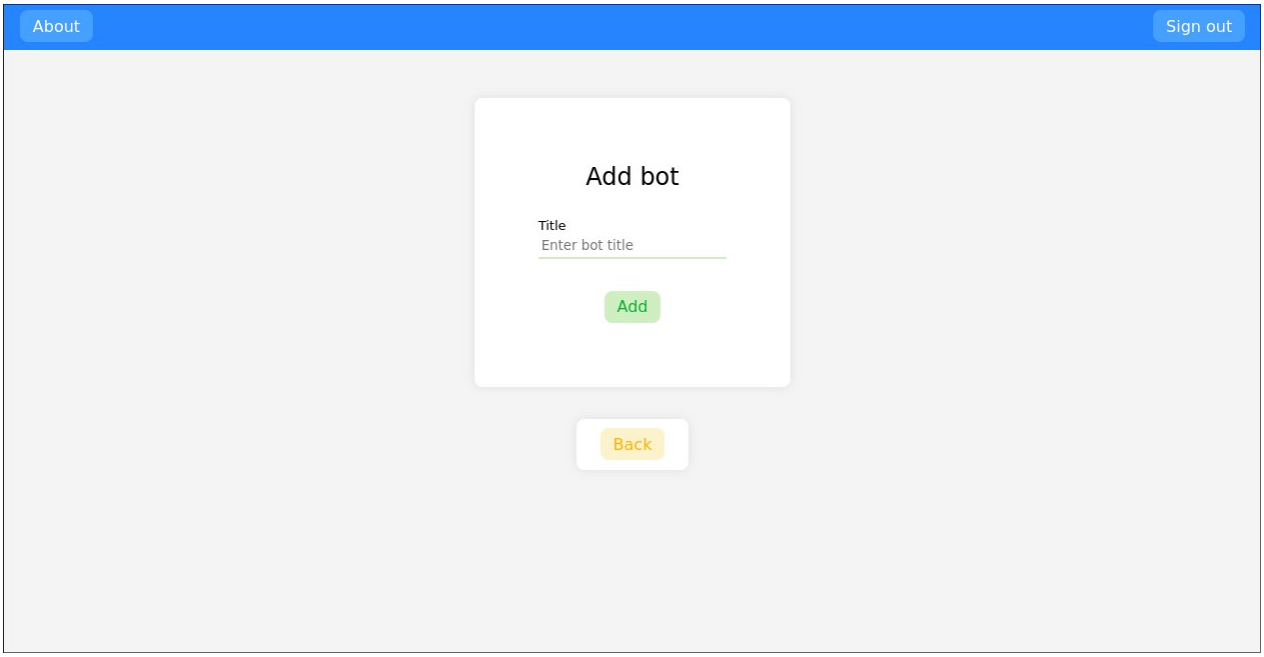

Рисунок 44 – Страница добавления бота

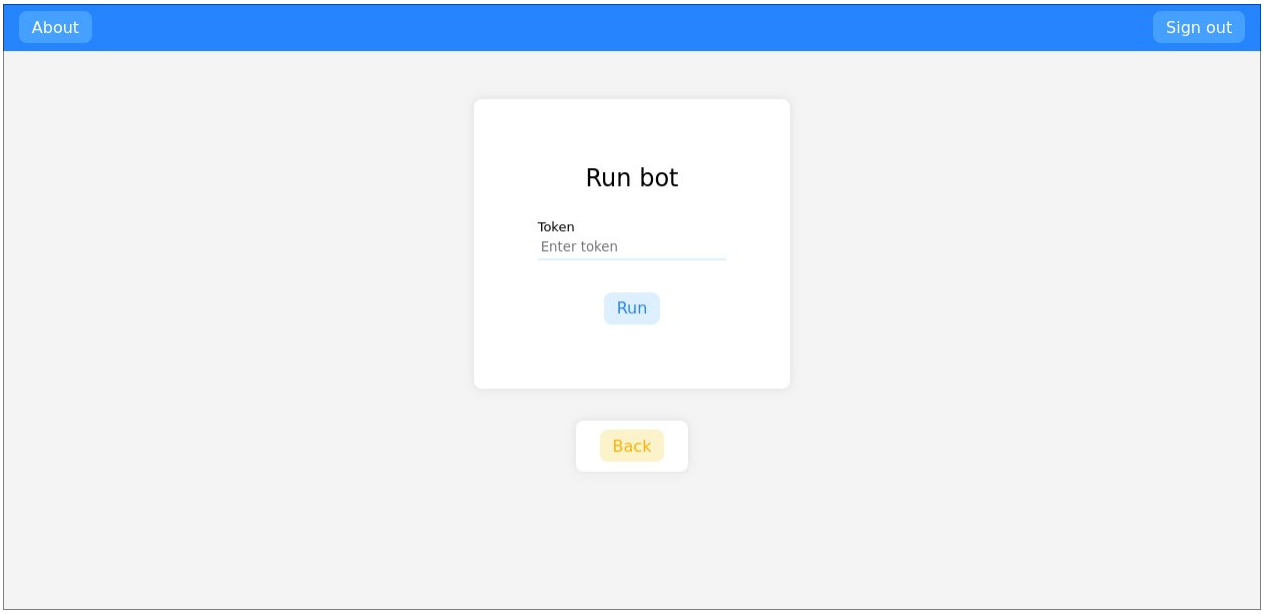

Рисунок 45 – Страница запуска бота

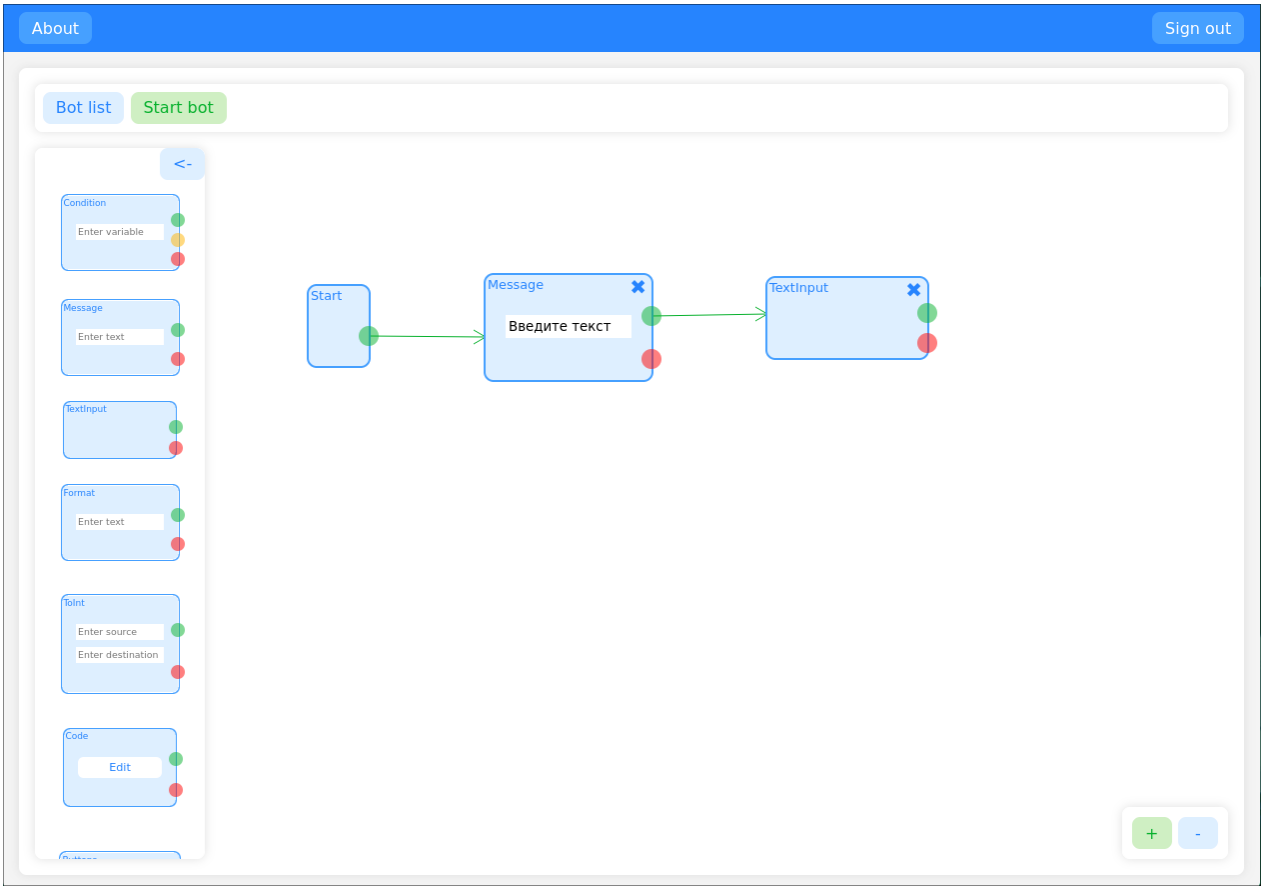

Рисунок 46 – Страница визуального редактора

3.4 Пример использования

Для демонстрации работы приложения был создан небольшой бот, который просит ввести пользователя код HTTP статуса ответа и после выводит соответствующую картинку с котом. Для получения картинок использовался сервис https://http.cat/. Экранные формы примера работы приложения представлены на рисунках 47-50.

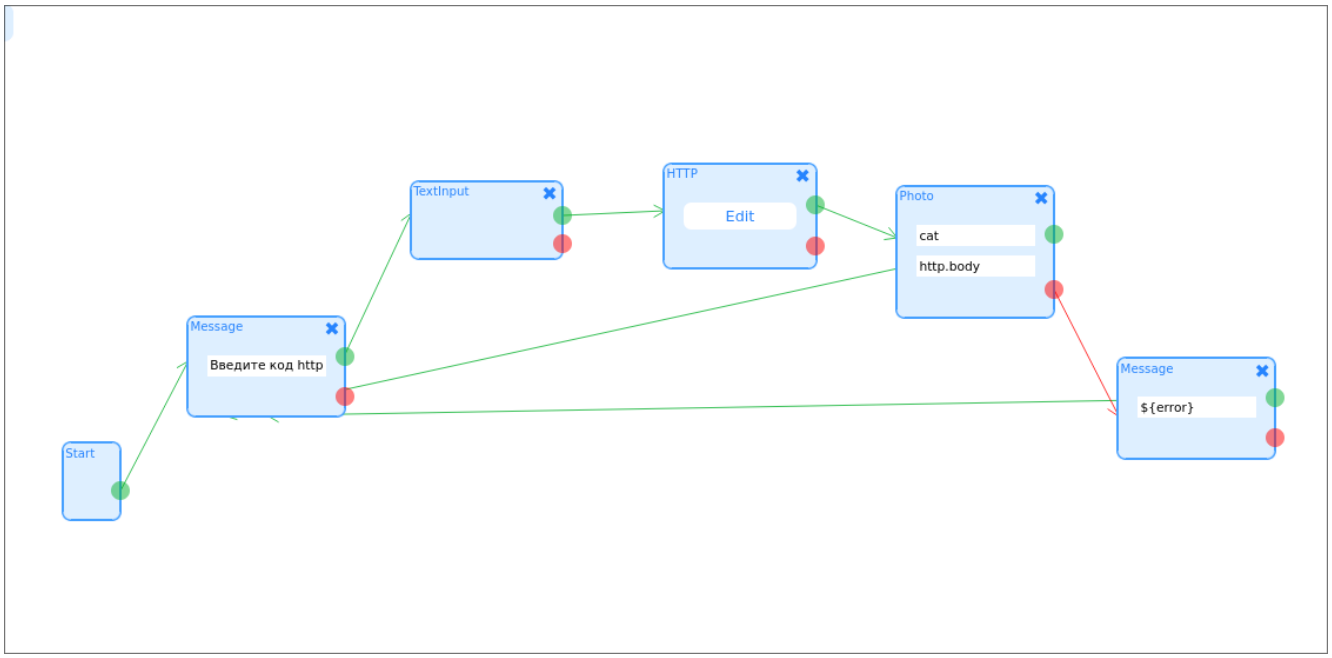

Рисунок 47 – Построение структуры бота

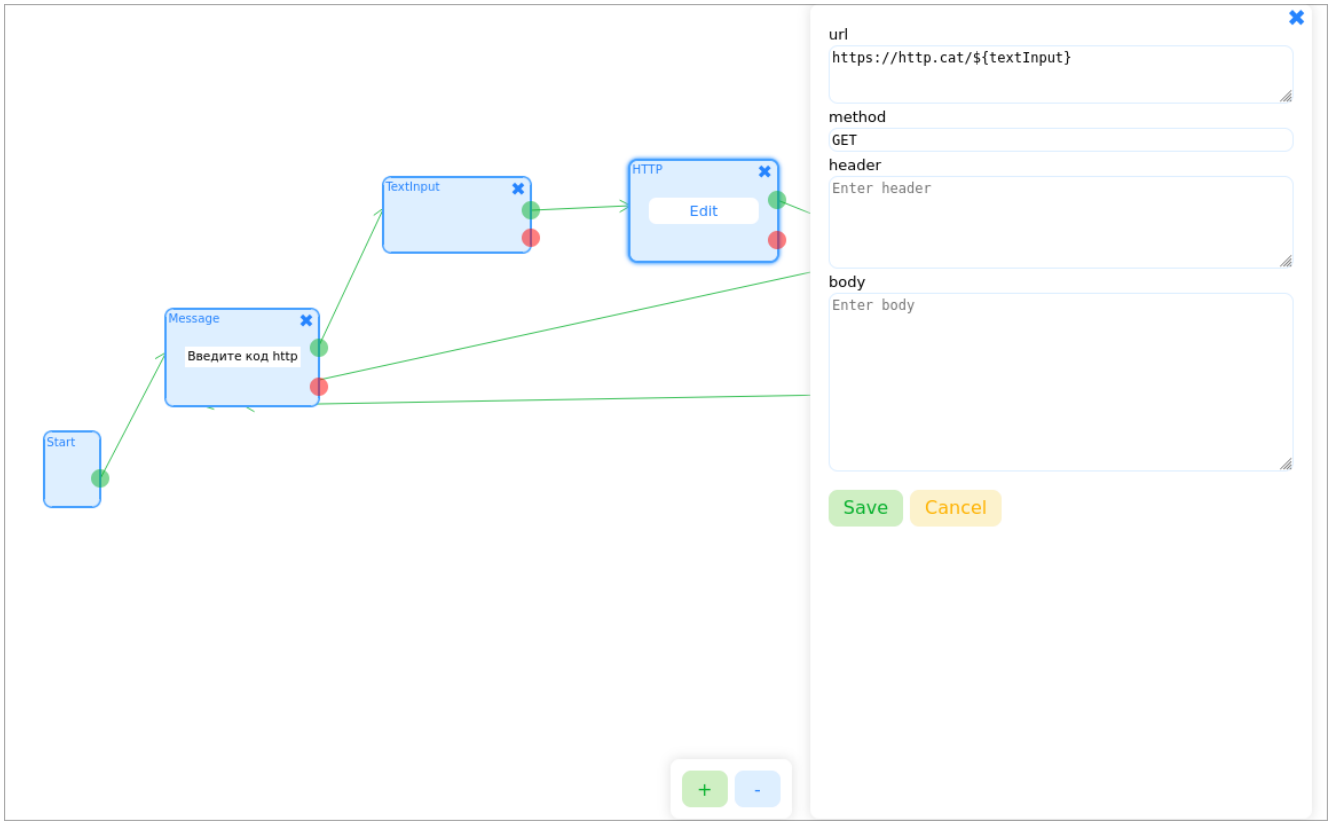

Рисунок 48 – Настройка HTTP компонента

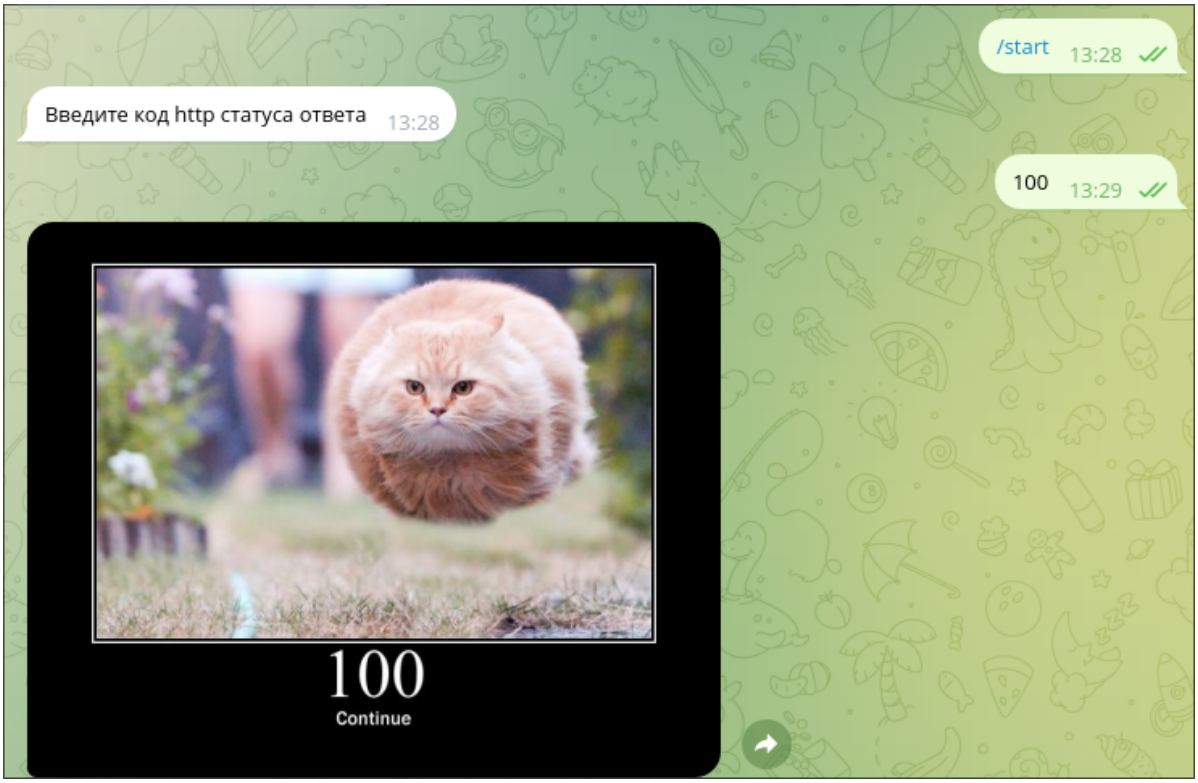

Рисунок 49 – Использование созданного бота

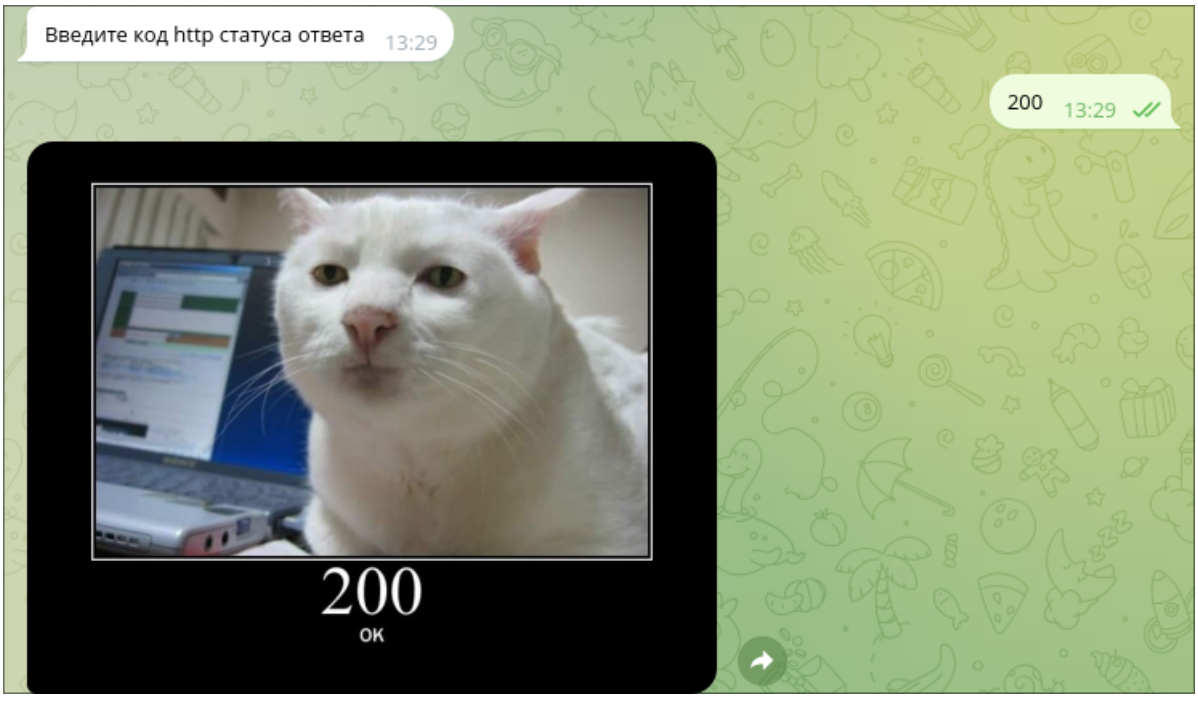

Рисунок 50 – Использование созданного бота

Выводы

В данном разделе были рассмотрены основные инструменты и средства, используемые при создании конструктора. Были рассмотрены аналоги, их плюсы и минусы.

Был выбран формат передаваемых данных между серверной и клиентской частями конструктора. У серверной части были выделены и разработаны программные интерфейсы, благодаря которым клиентская часть может обмениваться данными с сервисами.

На основе разработанных структур приложения, алгоритмов функционирования и диаграмм состояний была выполнена программная реализация конструктора. Листинг кода представлен в приложении Б.

Заключение

В ходе выполнения выпускной квалификационной работы был проведен анализ предметной области и обзор существующих аналогов конструктора Telegram-ботов. У рассмотренных аналогов были выявлены основные недостатки - ограниченность функционала при бесплатном использовании и показ рекламы. Также данные решения обладают закрытым исходным кодом, что не гарантирует приватность данных пользователей. Поэтому было принято решение о создании конструктора, который был бы лишен данных недостатков.

Из рассмотренных аналогов были выявлены требуемые функциональные возможности разрабатываемого продукта. Исходя из этих возможностей, было составлено расширенное техническое задание.

Была разработана общая структура конструктора, включающая в себя серверную и клиентскую части. Серверная часть обеспечивает выполнение основного функционала конструктора, а клиентская предоставляет пользователям удобный интерфейс для работы с ним. Серверная часть была разбита на ряд сервисов, которые инкапсулируют определенную часть функционала конструктора и предоставляют доступ к этим функциям через определенный программный интерфейс. Этими сервисами являются сервис пользователей, сервис ботов и сервис, обслуживающий ботов.

Для авторизации и реализации компонентов были выделены отдельные модули. Модуль компонентов был также разбит на подмодули, которые предоставляют интерфейсы для исполнения компонентов. С помощью реализации интерфейсов подмодуля ввода-вывода, возможно выполнение компонентов бота в разных мессенджерах.

Были определены алгоритмы обработки запросов пользователей сервиса ботов и пользователей бота. Алгоритм обработки запросов пользователей бота заключается в выполнении компонентов и сохранения текущего состояния контекста пользователя бота после обработки. Данный алгоритм обеспечивает работу ботов.

Интерфейс клиентской части был разбит на набор страниц, где находятся

основные визуальные объекты для взаимодействия с конструктором. Для содержимого каждой страницы был разработан свой шаблон.

Представлены модульная и компонентная структуры визуального редактора. Модульная структура позволила выделить определенные функциональные блоки редактора – набор структур и функций, которые ответственны за определенную часть редактора; компонентная структура помогла представить редактор как набор связанных компонентов в виде дерева.

Была разработана диаграмма состояний для клиентской части конструктора и редактора ботов, показывающая их поведение при действиях пользователя.

Для разработки были выбраны оптимальные инструменты и определен формат передаваемых данных, также был разработан интерфейс для обеспечения взаимодействия с сервисами, который работает по протоколу HTTP.

По разработанным структурам и алгоритмам был реализован конструктор Telegram-ботов. Исходный код разрабатываемого приложения выложен в открытый доступ на сервисе GitHub. Таким образом, каждый желающий может ознакомиться с ним и развернуть конструктор у себя, что предотвращает использование данных пользователей в нежелательных целях. Также пользователь сможет расширить возможности конструктора: добавить новые компоненты или ввести новые сервисы, которые будут обеспечивать работу ботов в других мессенджерах.

Приложение А

(обязательное)

Авторская справка

Я, Бушков Данил Андреевич, автор выпускной квалификационной работы «Разработка конструктора Telegram-ботов. Часть 1» сообщаю, что мне известно о персональной ответственности автора за разглашение сведений, подлежащих защите законами РФ о защите объектов интеллектуальной собственности.

Одновременно сообщаю, что:

1. При подготовке к защите выпускной квалификационной работы не использованы источники (документы, отчёты, диссертации, литература и т.п.), имеющие гриф секретности или «Для служебного пользования» ФГБОУ ВО «Вятский государственный университет» или другой организации.

2. Данная работа не связана с незавершёнными исследованиями или уже с завершёнными, но ещё официально не разрешёнными к опубликованию ФГБОУ ВО «Вятский государственный университет» или другими организациями.

3. Данная работа не содержит коммерческую информацию, способную нанести ущерб интеллектуальной собственности ФГБОУ ВО «Вятский государственный университет» или другой организации.

4. Данная работа не является результатом НИР или ОКР, выполняемой по договору с организацией.

5. В предлагаемом к опубликованию тексте нет данных по незащищённым объектам интеллектуальной собственности других авторов.

6. Использование моей дипломной работы в научных исследованиях оформляется в соответствии с законодательством РФ о защите интеллектуальной собственности отдельным договором.

Автор: Бушков Д. А. «____» _____ 2024 г. _____

подпись

Сведения по авторской справке подтверждаю: «____» _____2024 г.

Заведующий кафедрой ЭВМ: М. Л. Долженкова _____

подпись

Приложение Б

(обязательное)

Листинг кода

```
// Серверная часть

// Интерфейсы компонентов

package components

import (
        "encoding/json"

        "github.com/botscubes/bot-components/context"
        "github.com/botscubes/bot-components/io"
)

type Component interface {
        GetPath() string
        GetOutputs() Outputs
}
type Outputs interface {
        GetNextComponentId() *int64
        GetIdIfError() *int64
}

type (
        ActionComponent interface {
                Component

                Execute(ctx *context.Context) (*any, error)
        }

        ControlComponent interface {
                Component

                Execute(ctx *context.Context) error
        }

        InputComponent interface {
```

```
                Component

                Execute(ctx *context.Context, io io.IO) (*any, error)
        }

        OutputComponent interface {
                Component

                Execute(ctx *context.Context, io io.IO) error
        }
)

type ComponentTypeData struct {
        Type ComponentType `json:"type"`
}

type ComponentOutputs struct {
        NextComponentId *int64 `json:"nextComponentId"`
        IdIfError       *int64 `json:"idIfError"`
}

func (co *ComponentOutputs) GetNextComponentId() *int64 {
        return co.NextComponentId
}
func (co *ComponentOutputs) GetIdIfError() *int64 {
        return co.IdIfError
}

type ComponentData struct {
        ComponentTypeData
        Id   *int64 `json:"id"`
        Path string `json:"path"`
}

func (cd *ComponentData) GetPath() string {
        return cd.Path
}

func NewComponentFromJSON(tp ComponentType, jsonData []byte) (
   Component, error) {
        switch tp {
```

```
	case TypeStart:
		var s StartComponent
		err := json.Unmarshal(jsonData, &s)
		if err != nil {
			return nil, err
		}
		return &s, nil
	case TypeFormat:
		var f FormatComponent
		err := json.Unmarshal(jsonData, &f)
		if err != nil {
			return nil, err
		}
		return &f, nil
	case TypeCondition:
		var c ConditionComponent
		err := json.Unmarshal(jsonData, &c)
		if err != nil {
			return nil, err
		}
		return &c, err

	case TypeMessage:
		var m MessageComponent
		err := json.Unmarshal(jsonData, &m)
		if err != nil {
			return nil, err
		}
		return &m, err
	case TypeTextInput:
		var ti TextInputComponent
		err := json.Unmarshal(jsonData, &ti)
		if err != nil {
			return nil, err
		}
		return &ti, err
	case TypeButtons:
		var bc ButtonComponent
		err := json.Unmarshal(jsonData, &bc)
		if err != nil {
			return nil, err
```

```
                }
                return &bc, err
        default:
                return nil, ErrComponentTypeNotExist
        }
}


// Реализация компонентов

package components

import (
        "github.com/botscubes/bot-components/context"
        "github.com/botscubes/bot-components/format"
        "github.com/botscubes/bot-components/io"
)

type MessageComponent struct {
        ComponentData

        Outputs ComponentOutputs `json:"outputs"`
        Data    struct {
                Text string `json:"text"`
        } `json:"data"`
}

func (mc *MessageComponent) GetOutputs() Outputs {
        return &mc.Outputs
}
func (mc *MessageComponent) Execute(ctx *context.Context, io io.IO)
   error {
        var s string
        s, err := format.Format(mc.Data.Text, ctx)
        if err != nil {
                return err
        }
        io.PrintText(s)
        return nil
}
```

```go
package components

import (
	"github.com/botscubes/bot-components/context"
	"github.com/botscubes/bot-components/format"
)

type FormatComponent struct {
	ComponentData

	Outputs ComponentOutputs `json:"outputs"`
	Data    struct {
		FormatString string `json:"formatString"`
	} `json:"data"`
}

func (fc *FormatComponent) GetOutputs() Outputs {
	return &fc.Outputs
}

func (fc *FormatComponent) Execute(ctx *context.Context) (*any, error
   ) {
	var s any
	s, err := format.Format(fc.Data.FormatString, ctx)
	if err != nil {
		return nil, err
	}
	return &s, nil
}

package components

import (
	"errors"

	"github.com/botscubes/bot-components/context"
	"github.com/botscubes/bot-components/io"
)

type TextInputComponent struct {
	ComponentData
```

```
        Outputs ComponentOutputs `json:"outputs"`
}

func (tc *TextInputComponent) GetOutputs() Outputs {
        return &tc.Outputs
}

func (tc *TextInputComponent) Execute(ctx *context.Context, io io.IO)
    (*any, error) {
        s := io.ReadText()
        if s == nil {
                tc.Outputs.NextComponentId = tc.Id
                return nil, nil
        }
        if *s == "" {
                return nil, errors.New("Empty␣string␣entered")
        }
        var a any
        a = *s
        return &a, nil
}

\\ Реализация исполнителя

package exec

import (
        "github.com/botscubes/bot-components/components"
        "github.com/botscubes/bot-components/context"
        "github.com/botscubes/bot-components/io"
)

type Executor struct {
        io  io.IO
        ctx *context.Context
}

func NewExecutor(ctx *context.Context, io io.IO) *Executor {
        return &Executor{
                io,
                ctx,
```

```
        }
}

func (e *Executor) Execute(component components.Component) (*int64,
   error) {
        switch cmp := component.(type) {
        case components.ActionComponent:

                v, err := cmp.Execute(e.ctx)
                if err != nil {
                        return cmp.GetOutputs().GetIdIfError(), err
                }
                e.ctx.SetValue(component.GetPath(), v)
        case components.ControlComponent:
                err := cmp.Execute(e.ctx)
                if err != nil {
                        return cmp.GetOutputs().GetIdIfError(), err
                }
        case components.InputComponent:
                v, err := cmp.Execute(e.ctx, e.io)
                if err != nil {
                        return cmp.GetOutputs().GetIdIfError(), err
                }
                if v != nil {
                        e.ctx.SetValue(component.GetPath(), v)
                }
        case components.OutputComponent:
                err := cmp.Execute(e.ctx, e.io)
                if err != nil {
                        return cmp.GetOutputs().GetIdIfError(), err
                }

        default:
                return nil, ErrComponentNotImplInterface
        }

        nextId := component.GetOutputs().GetNextComponentId()

        return nextId, nil
}
```

```
# Регистрация обработчиков сервиса ботов

package app

import (
        "github.com/botscubes/bot-service/internal/api/handlers"
        m "github.com/botscubes/bot-service/internal/api/middlewares"
        "github.com/gofiber/fiber/v2"
        "github.com/gofiber/fiber/v2/middleware/recover"
)

func (app *App) regiterHandlers(h *handlers.ApiHandler) {
        app.server.Get("/api/bots/health", handlers.Health)

        // panic recover
        app.server.Use(recover.New())

        // Auth middleware
        app.server.Use(m.Auth(&app.sessionStorage, &app.conf.JWTKey,
   app.log))

        api := app.server.Group("/api")
        bots := api.Group("/bots")
        bot := bots.Group("/:botId<int>", m.GetBotMiddleware(app.db,
   app.log))
        groups := bot.Group("/groups")
        group := groups.Group("/:groupId<int>", m.GetGroupMiddleware(
   app.db, app.log))
        components := group.Group("/components")
        component := components.Group("/:componentId<int>", m.
   GetComponentMiddleware(app.db, app.log))

        regBotsHandlers(bots, h)
        regBotHandlers(bot, h)
        regGroupHandlers(group, h)
        regComponentsHandlers(components, h)

        regComponentHandlers(component, h)

        // custom 404 handler
        app.server.Use(handlers.NotFoundHandler)
```

```
}

// Bot handlers
func regBotsHandlers(bots fiber.Router, h *handlers.ApiHandler) {
        // Create new bot
        bots.Post("", h.NewBot)
        // Get user bots
        bots.Get("", h.GetBots)

}

func regBotHandlers(bot fiber.Router, h *handlers.ApiHandler) {

        // Delete bot
        bot.Delete("", h.DeleteBot)

        // Set bot token
        bot.Patch("/token", h.SetBotToken)

        // Delete bot token
        bot.Delete("/token", h.DeleteBotToken)

        bot.Get("/token", h.GetBotToken)
        // Start bot
        bot.Patch("/start", h.StartBot)

        // Stop bot
        bot.Patch("/stop", h.StopBot)

        bot.Get("/status", h.GetBotStatus)
}

func regGroupsHandlers(groups fiber.Router, h *handlers.ApiHandler) {

}

func regGroupHandlers(group fiber.Router, h *handlers.ApiHandler) {
        group.Post("/connections", h.AddConnetion)
        group.Delete("/connections", h.DeleteConnection)
}
```

```
func regComponentsHandlers(components fiber.Router, h *handlers.
   ApiHandler) {

        // Get bot components
        components.Get("", h.GetBotComponents)
        components.Post("", h.AddComponent)

}

func regComponentHandlers(component fiber.Router, h *handlers.
   ApiHandler) {
        component.Delete("", h.DeleteComponent)
        component.Patch("/position", h.SetComponentPosition)
        component.Patch("/data", h.UpdateComponentData)
}

# Пример обработчика сервиса ботов

func (h *ApiHandler) AddComponent(ctx *fiber.Ctx) error {
        botId, ok := ctx.Locals("botId").(int64)
        if !ok {
                h.log.Errorw("BotId␣to␣int64␣convert", "error",
   ErrUserIDConvertation)
                return ctx.SendStatus(fiber.StatusInternalServerError
   )
        }
        groupId, ok := ctx.Locals("groupId").(int64)
        if !ok {
                h.log.Errorw("GroupId␣to␣int64␣convert", "error",
   ErrUserIDConvertation)
                return ctx.SendStatus(fiber.StatusInternalServerError
   )
        }

        reqData := new(model.AddComponentReq)
        if err := ctx.BodyParser(reqData); err != nil {
                return ctx.SendStatus(fiber.StatusBadRequest)
        }

        if errValidate := reqData.Validate(); errValidate != nil {
                return ctx.Status(fiber.StatusUnprocessableEntity).
```

```
   JSON(errValidate)
        }

        compId, err := h.db.AddComponent(botId, groupId, &model.
   Component{
                Position: reqData.Position,
                ComponentData: components.ComponentData{
                        ComponentTypeData: components.
   ComponentTypeData{
                                Type: reqData.Type,
                        },
                        Path: reqData.Type,
                },
        })
        if err != nil {
                h.log.Errorw("failed␣add␣component", "error", err)
                return ctx.SendStatus(fiber.StatusInternalServerError
   )
        }

        dataRes := &AddComponentRes{
                Id: compId,
        }
        return ctx.Status(fiber.StatusCreated).JSON(dataRes)
}

// Обработчики сервиса пользователя

package server

import (
        "context"
        "net/http"
        "time"

        "github.com/botscubes/user-service/internal/user"
        "github.com/botscubes/user-service/pkg/jwt"
        "github.com/botscubes/user-service/pkg/password_hash"
        //"github.com/botscubes/user-service/pkg/service_error"
        "github.com/labstack/echo/v4"
)
```

```
type ResponseToken struct {
        Token string `json:"token"`
        //Error *service_error.ServiceError `json:"error"`
}

// Handlers for server. Handlers are implemented using a closure.
func (s *Server) bindHandlers() {

        s.echo.POST("/api/users/signup", func(c echo.Context) error {
                var u *user.User = new(user.User)
                if err := c.Bind(u); err != nil {
                        return c.JSON(http.StatusBadRequest, nil)
                }

                u, service_err := user.NewUser(u.Login, u.Password)
                if service_err != nil {
                        return c.JSON(http.StatusUnprocessableEntity,
    service_err)
                }

                if exists, err := s.userModel.LoginExists(context.
   Background(), u.Login); err != nil {

                        s.echo.Logger.Error(err)
                        return c.NoContent(http.
   StatusInternalServerError)
                } else if exists {
                        return c.JSON(http.StatusUnprocessableEntity,
    user.ErrLoginExists)
                }

                var err error = nil
                u.Password, err = password_hash.GetPasswordHash(u.
   Password, s.conf.Server.Salt)
                if err != nil {
                        s.echo.Logger.Error(err)
                        return c.NoContent(http.
   StatusInternalServerError)
                }
```

```
                err = s.userModel.SaveUser(context.Background(), u)
                if err != nil {
                        s.echo.Logger.Error(err)
                        return c.NoContent(http.
    StatusInternalServerError)
                }

                return c.NoContent(http.StatusCreated)
        })

        s.echo.POST("/api/users/signin", func(c echo.Context) error {
                var u *user.User = new(user.User)
                if err := c.Bind(u); err != nil {
                        return c.NoContent(http.StatusBadRequest)
                }

                u, service_err := user.NewUser(u.Login, u.Password)
                if service_err != nil {
                        return c.JSON(http.StatusUnprocessableEntity,
     service_err)
                }

                id, password, err := s.userModel.
    GetIdAndPasswordByLogin(context.Background(), u.Login)
                if err != nil {

                        s.echo.Logger.Error(err)
                        return c.NoContent(http.
    StatusInternalServerError)
                }
                if id == 0 {
                        return c.JSON(http.StatusUnprocessableEntity,
     user.ErrLoginNotExists)
                }

                if !password_hash.CheckPasswordHash(u.Password,
    password, s.conf.Server.Salt) {
                        return c.JSON(http.StatusUnprocessableEntity,
     user.ErrPasswordIsNotEqual)
                }
                claims := jwt.NewUserClaims(
```

```
                        id,
                        time.Duration(s.conf.Server.TokenLifeTime)*
time.Second,
                )
                token, err := jwt.GenerateToken(
                        claims,
                        s.conf.Server.JWTKey,
                )
                if err != nil {
                        s.echo.Logger.Error(err)
                        return c.NoContent(http.
StatusInternalServerError)
                }

                err = s.tokenStorage.SaveToken(
                        context.Background(),
                        token,
                        claims.GetLifeDuration(),
                )
                if err != nil {

                        s.echo.Logger.Error(err)
                        return c.NoContent(http.
StatusInternalServerError)
                }

                return c.JSON(http.StatusCreated, ResponseToken{token
})
        })

        s.echo.DELETE("/api/users/signout", func(c echo.Context)
error {
                token := c.Get("token").(string)
                if token == "" {
                        return c.NoContent(http.StatusNoContent)
                }
                err := s.tokenStorage.DeleteToken(context.Background
(), token)
                if err != nil {
                        s.echo.Logger.Error(err)
                        return c.NoContent(http.
```

```
   StatusInternalServerError)
                }

                return c.NoContent(http.StatusNoContent)

        }, JWT(s.conf.Server.JWTKey, s.tokenStorage, s.echo.Logger))
}

// Цикл выполнения компонентов в сервисе, обслуживающем ботов

package bot

import (
        "errors"

        "github.com/botscubes/bot-components/context"
        "github.com/botscubes/bot-components/exec"
        "github.com/botscubes/bot-components/io"
        "github.com/botscubes/bot-worker/internal/config"
)

func (bw *BotWorker) execute(botId int64, groupId int64, userId int64
   , io io.IO, step int64, ctx *context.Context) error {
        components, err := bw.storage.components(botId, groupId)
        if err != nil {
                return err
        }
        const MAX_VISIT = config.MaxLoopInExecution
        visitedComponents := make(map[int64]int64)
        e := exec.NewExecutor(ctx, io)
        for {
                _, ok := visitedComponents[step]
                if ok {
                        visitedComponents[step]++
                } else {
                        visitedComponents[step] = 1
                }
                if visitedComponents[step] > MAX_VISIT {

                        return errors.New("Loop too long")
                }
```

```
                componentData, ok := components[step]
                if !ok {
                        break
                }

                bw.log.Debug(string(componentData.Data))
                component, err := componentData.Component()
                if err != nil {
                        return err
                }
                st, err := e.Execute(component)
                if err != nil {
                        var val any = err.Error()
                        ctx.SetValue("error", &val)
                }
                if st == nil {
                        step = 0
                        break
                }
                if step == *st {
                        break
                }

                step = *st
        }

        err = bw.storage.setUserStep(botId, groupId, userId, step)
        if err != nil {
                return err
        }
        err = bw.storage.setContext(botId, groupId, userId, ctx)
        if err != nil {
                return err
        }
        return nil
}

// Клиентская часть

// Компонент редактора
```

```
import { For, JSX, Show, createEffect, createSignal, onMount } from "
   solid-js";
import { createStore } from "solid-js/store";
import {
  handleMouseMove,
  getDeleteComponentHandler,
  getAddSelectedComponentHandler,
  getSelectComponentHandler,
  getFinishConnectionHandler,
  handleMouseUp,
  handleMouseDown,
  getDeleteConnectionHandler,
} from "./eventHandlers";
import { Component } from "./components/Component";
import EditorController from "./EditorController";

import "./Editor.css";

import { ExtendedComponentData } from "./EditorController/
   EditorStorage/ComponentStorage/types";
import { SpecificComponentCreator } from "./EditorController/
   SpecificComponent";

import { ConditionComponentCreator } from "./EditorController/
   components/ConditionComponent";
import { Line, LinePosition } from "./components/Line";
import { BotStatus, EditorProps, LineData } from "./types";
import { EditorClient } from "./EditorController/api/EditorClient";
import { A, useNavigate } from "@solidjs/router";
import { MessageComponentCreator } from "./EditorController/
   components/MessageComponent";
import { TextInputComponentCreator } from "./EditorController/
   components/TextInputComponent";
import { FormatComponentCreator } from "./EditorController/components
   /FormatComponent";
import { ButtonComponentCreator } from "./EditorController/components
   /ButtonComponent";
import BotClient from "~/api/bot/BotClient";

export default function Editor(props: EditorProps) {
```

```
const [componentStore, setComponentStore] = createStore<
  Record<number, ExtendedComponentData>
>({});
const [addingComponentContent, setAddingComponentContent] =
 createSignal<
  (() => JSX.Element) | undefined
>(undefined);
const [addingComponentPosition, setAddingComponentPosition] =
 createSignal({
  x: 0,
  y: 0,
});
const [userSelect, setUserSelect] = createSignal(true);
const [scale, setScale] = createSignal(1);
const [line, setLine] = createSignal<LineData | undefined>(
 undefined);
const [lines, setLines] = createStore<Record<string, LineData>>({})
 ;
const [loading, setLoading] = createSignal(false);
const [error, setError] = createSignal<Error | undefined>();
const [errors, setErrors] = createSignal<Array<Error>>([]);
const [botStatus, setBotStatus] = createSignal(BotStatus.Stopped);
createEffect(() => {
  if (error()) {
    const err = error() as Error;
    setErrors((errors) => [...errors, err]);

    setTimeout(() => {
      setErrors((errors) => errors.filter((error) => error != err))
 ;
    }, 5000);
  }
});

// eslint-disable-next-line solid/reactivity
const logger = props.logger;

const navigate = useNavigate();
const editor: EditorController = new EditorController(
  {
    componentStore: [componentStore, setComponentStore],
```

```
    addingComponent: {
      setContent: setAddingComponentContent,
      setPosition: setAddingComponentPosition,
    },
    setUserSelect: setUserSelect,
    scale: {
      get: scale,
      set: setScale,
    },
    line: {
      set: setLine,
    },
    lineStore: [lines, setLines],
    setLoading: setLoading,
    navigate: navigate,
    error: {
      set: setError,
    },
    bot: {
      status: {
        set: setBotStatus,
        get: botStatus,
      },
    },
  },
  // eslint-disable-next-line solid/reactivity
  new EditorClient(props.httpClient, props.token, props.botId, 1),
  logger
);
onMount(() => {
  editor.init();
});

const [showComponentSelection, setShowComponentSelection] =
  createSignal(false);
const componentCreatorList: Array<SpecificComponentCreator> = [
  new ConditionComponentCreator(editor),
  new MessageComponentCreator(editor),
  new TextInputComponentCreator(editor),
  new FormatComponentCreator(editor),
  new ButtonComponentCreator(editor),
```

```
];
return (
  <div
    id="editor-area"
    ref={(editorArea) => {
      editor.setEditorArea(editorArea);
    }}
    data-editor-area
    onMouseMove={[handleMouseMove, editor]}
    onMouseUp={[handleMouseUp, editor]}
    onMouseDown={[handleMouseDown, editor]}
    style={{
      "user-select": !userSelect() ? "none" : undefined,
    }}
  >
    <div class="fixed-area no-events">
      <div class="editor-errors">
        <For each={errors()}>
          {(error) => (
            <div class="editor-error">
              <div class="error">{error.message}</div>
            </div>
          )}
        </For>
      </div>

      <div class="scale-buttons events">
        <button class="green-button" onClick={() => editor.zoomIn()
}>
          +
        </button>
        <div class="separator"> </div>
        <button class="blue-button" onClick={() => editor.zoomOut()
}>
          {" "}
          -{" "}
        </button>
      </div>
      <div id="editor-menu-panel" class="events">
        <A href="/bots" class="blue-button">
          Bot list
```

```
          </A>
          <div class="separator"> </div>
          <Show
            when={botStatus() == BotStatus.Stopped}
            fallback={
              <button onClick={() => editor.stopBot()} class="yellow-
    button">
                Stop bot
              </button>
            }
          >
            <A
              href={"/bots/" + props.botId + "/start?prev=edit"}
              class="green-button"
            >
              Start bot
            </A>
          </Show>
          <div class="separator"> </div>
          <Show when={loading()}>
            <div class="loading">Loading...</div>
          </Show>
        </div>
        <Show
          when={showComponentSelection()}
          fallback={
            <button
              class="events blue-button show-panel-btn"
              onClick={() => setShowComponentSelection(true)}
            >
              {"->"}
            </button>
          }
        >
          <div id="component-selection-panel" class="events">
            <button
              class="hide-panel-btn blue-button"
              onClick={() => {
                setShowComponentSelection(false);
              }}
            >
```

```
            {"<-"}
          </button>
          <div class="component-list">
            <For each={componentCreatorList}>
              {(creator) => (
                <div
                  class="component reduce"
                  onMouseDown={(event: MouseEvent) => {
                    editor.startAddingComponent(event, creator);
                  }}
                >
                  <div class="no-events">{creator.content()}</div>
                </div>
              )}
            </For>
          </div>
        </div>
      </Show>
      <Show when={addingComponentContent()}>
        <div
          class="scaling"
          style={{
            transform: `scale(${scale()})`,
          }}
        >
          <div
            class="component absolute adding-component"
            style={{
              top: addingComponentPosition().y + "px",
              left: addingComponentPosition().x + "px",
            }}
          >
            {addingComponentContent()?.()}
          </div>
        </div>
      </Show>
    </div>
    <div
      class="scaling"
      style={{
        transform: `scale(${scale()})`,
```

```
      }}
    >
      <For each={Object.values(componentStore)}>
        {(component) => {
          const componentStyle = {
            width: 100, //px
            connectionPointSize: 20, //px
          };
          return (
            <Component
              scale={scale()} //editor.getEditorData().scale}
              componentData={component}
              componentStyle={componentStyle}
              deleteComponent={getDeleteComponentHandler(editor)}
              selectComponent={getSelectComponentHandler(editor)}
              addSelectedComponent={getAddSelectedComponentHandler(
editor)}
              finishConnection={getFinishConnectionHandler(editor)}
              deleteConnection={getDeleteConnectionHandler(editor)}
              //moveConnection={handleMoveConnection}
              //moveCommandConnection={handleMoveCommandConnection}
            >
              {component.content()}
            </Component>
          );
        }}
      </For>

      <Show when={line()}>
        <Line position={line()?.position!} color={line()?.color ??
"black"} />
      </Show>
      <For each={Object.values(lines)}>
        {(line) => (
          <Line position={line.position} color={line.color ?? "
black"} />
        )}
      </For>
    </div>
  </div>
);
```

```
}

// Контроллер редактора

import { BotStatus, EditorData } from "../types";
import { Position } from "../shared/types";
import type EditorState from "./EditorState";
import WaitingState from "./states/WaitingState";
import { getRelativeMousePosition } from "./halpers/mouse";
import ComponentController from "./ComponentController";
import Logger from "~/logging/Logger";
import AddingComponentState from "./states/AddingComponentState";
import {
  SpecificComponent,
  SpecificComponentCreator,
} from "./SpecificComponent";
import ConnectionController from "./ConnectionController";
import LineStore from "./EditorStorage/LineStore";
import ConnectionState from "./states/ConnectionState";
import { HTTPResponse, checkPromise } from "~/api/HTTPResponse";
import { EditorClient } from "./api/EditorClient";
import ComponentStore from "./EditorStorage/ComponentStorage/
   ComponentStore";
import { StartComponentController } from "./components/StartComponent
   ";
import { APIComponentData, APIComponentType } from "./api/types";
import { StartContent } from "../components/ComponentContent/contents
   /StartContent";
import { ConditionComponentController } from "./components/
   ConditionComponent";
import { ConditionContent } from "../components/ComponentContent/
   contents/ConditionContent";
import { MessageContent } from "../components/ComponentContent/
   contents/MessageContent";
import { MessageComponentController } from "./components/
   MessageComponent";
import { TextInputComponentController } from "./components/
   TextInputComponent";
import { TextInputContent } from "../components/ComponentContent/
   contents/TextInputContent";
import { FormatComponentController } from "./components/
```

```
   FormatComponent";
import { FormatContent } from "../components/ComponentContent/
   contents/FormatContent";
import { ButtonComponentController } from "./components/
   ButtonComponent";
import { ButtonContent } from "../components/ComponentContent/
   contents/ButtonContent";
import BotClient from "~/api/bot/BotClient";

export default class EditorController {
  private readonly zoomSize = 0.05;
  private editorState: EditorState = new WaitingState(this);
  private editorArea?: HTMLElement;
  private _components: ComponentController;
  private _connections: ConnectionController;

  constructor(
    private editor: EditorData,
    private _client: EditorClient,
    private logger: Logger
  ) {
    const componentStore = new ComponentStore(...editor.
   componentStore);
    this._components = new ComponentController(this, componentStore,
   logger);
    this._connections = new ConnectionController(
      this,
      new LineStore(...editor.lineStore),
      componentStore
    );
  }
  get client() {
    return this._client;
  }
  get connections() {
    return this._connections;
  }
  get components() {
    return this._components;
  }
  get state() {
```

```
    return this.editorState;
  }
  get addingComponent() {
    return this.editor.addingComponent;
  }
  get area() {
    return this.editorArea;
  }
  get setUserSelect() {
    return this.editor.setUserSelect;
  }
  get line() {
    return this.editor.line;
  }

  get error() {
    return this.editor.error;
  }

  async init() {
    this.getBotStatus();
    const [components, error] = await this.httpRequest(() =>
      this._client.getComponents()
    );
    if (error) {
      this.editor.error.set(error);
      return;
    }
    if (components) {
      for (const component of components) {
        let abitityToDelete = true;
        if (component.type == APIComponentType.Start) {
          abitityToDelete = false;
        }
        const specificComponent = this.createSpecificComponent(
 component);
        if (specificComponent) {
          this.components.add(
            component.id,
            component.position,
            specificComponent,
```

```
          {},
          abitityToDelete
        );
        for (const [_, value] of Object.entries(component.
 connectionPoints)) {
          this.components.addConnectionPoint(
            component.id,
            value.sourceComponentId,
            value.sourcePointName,
            value.relativePointPosition
          );
        }
      } else {
        this.editor.error.set(
          new Error(
            `This type of component does not exist: ${component.
 type}`
          )
        );
        return;
      }
    }
  }
  this.connections.addLinesBetweenComponents();
}

selectComponent(id: number, mousePosition: Position) {
  this.editorState.selectComponent(id, mousePosition);
}

setEditorArea(editorArea?: HTMLElement) {
  this.editorArea = editorArea;
}

startConnection(
  componentId: number,
  pointId: string,
  clientPosition: Position,
  pointColor: string
) {
  this.editorState.startConnection(
```

```
      componentId,
      pointId,
      this.getRelativeMousePosition(clientPosition),
      pointColor
    );
  }
  finishConnection(
    componentId: number,
    connectionPosition: Position,
    relativePointPosition: Position
  ) {
    this.editorState.finishConnection(
      componentId,
      connectionPosition,
      relativePointPosition
    );
  }
  deleteConnection(
    targetComponentId: number,
    sourceComponentId: number,
    sourcePointId: string,
    clientPosition: Position
  ) {
    const line = this.connections.getLine(sourceComponentId,
   sourcePointId);

    this.connections.delete(
      targetComponentId,
      sourceComponentId,
      sourcePointId
    );

    this.setState(
      new ConnectionState(
        this,
        {
          componentId: sourceComponentId,
          pointId: sourcePointId,
          pointPosition: line.position.start,
        },
        this.getRelativeMousePosition(clientPosition),
```

```
      line.color
    )
  );
}
handleMouseDown(event: MouseEvent) {
  this.editorState.handleMouseDown(event);
}
handleMouseMove(event: MouseEvent) {
  this.editorState.handleMouseMove(event);
}
handleMouseUp(event: MouseEvent) {
  this.editorState.handleMouseUp(event);
}
setState(state: EditorState) {
  this.logger.info("Editor: state changed to " + state.name);
  this.editorState = state;
}

getRelativeMousePosition(mousePosition: Position): Position {
  let relativeMousePosition = { x: 0, y: 0 };
  if (this.editorArea) {
    relativeMousePosition = getRelativeMousePosition(
      this.editorArea,
      mousePosition,
      this.editor.scale.get()
    );
  }
  return relativeMousePosition;
}

startAddingComponent(event: MouseEvent, creator:
 SpecificComponentCreator) {
  this.setState(new AddingComponentState(this, event, creator));
}
zoomIn() {
  this.editor.scale.set((scale) => scale + this.zoomSize);
}
zoomOut() {
  this.editor.scale.set((scale) => scale - this.zoomSize);
}
private async getBotStatus() {
```

```
    const [status, error] = await this.httpRequest(() =>
      this._client.getBotStatus()
    );
    if (error) {
      this.editor.error.set(error);
      return;
    }
    if (status == 1) {
      this.editor.bot.status.set(BotStatus.Running);
    } else {
      this.editor.bot.status.set(BotStatus.Stopped);
    }
  }
  async stopBot() {
    const [_, error] = await this.httpRequest(() => this._client.
   stopBot());
    if (error) {
      this.editor.error.set(error);
      return;
    }
    this.editor.bot.status.set(BotStatus.Stopped);
  }

  async httpRequest<T>(
    request: () => Promise<HTTPResponse<T>>
  ): Promise<[T | undefined, Error | undefined]> {
    this.editor.setLoading(true);
    try {
      const response = await checkPromise(request(), this.logger);
      if (response.statusUnauthorized()) {
        this.editor.navigate("/signin");
      }
      response.check();
      return [response.data, undefined];
    } catch (e) {
      return [undefined, e as Error];
    } finally {
      this.editor.setLoading(false);
    }
  }
  createSpecificComponent(
```

```
  component: APIComponentData
): SpecificComponent | undefined {
  switch (component.type) {
    case APIComponentType.Start: {
      const controller = new StartComponentController(this,
component.id);

      return [
        controller,
        () => (
          <StartContent
            outputs={component.outputs}
            handlers={controller.getHandlers()}
          />
        ),
      ];
    }
    case APIComponentType.Condition: {
      const controller = new ConditionComponentController(this,
component.id);

      return [
        controller,
        () => (
          <ConditionContent
            data={component.data}
            outputs={component.outputs}
            handlers={controller.getHandlers()}
          />
        ),
      ];
    }
    case APIComponentType.Message: {
      const controller = new MessageComponentController(this,
component.id);

      return [
        controller,
        () => (
          <MessageContent
            data={component.data}
```

```
            outputs={component.outputs}
            handlers={controller.getHandlers()}
          />
        ),
      ];
    }
    case APIComponentType.TextInput: {
      const controller = new TextInputComponentController(this,
component.id);

      return [
        controller,
        () => (
          <TextInputContent
            data={component.data}
            outputs={component.outputs}
            handlers={controller.getHandlers()}
          />
        ),
      ];
    }
    case APIComponentType.Format: {
      const controller = new FormatComponentController(this,
component.id);

      return [
        controller,
        () => (
          <FormatContent
            data={component.data}
            outputs={component.outputs}
            handlers={controller.getHandlers()}
          />
        ),
      ];
    }
    case APIComponentType.Buttons: {
      const controller = new ButtonComponentController(
        this,
        component.id,
        component.data.buttons,
```

```
          component.outputs
        );

        return [
          controller,
          () => (
            <ButtonContent
              data={{
                text: component.data.text,
                buttons: controller.buttons(),
              }}
              handlers={controller.getHandlers()}
              abilityToAdd={controller.abilityToAdd()}
            />
          ),
        ];
      }
      default: {
        return undefined;
      }
    }
  }
}
```

Приложение В

(справочное)

Список сокращений и обозначений

СУБД - Система управления базами данных

API - Application Programming Interface

CSS - Cascading Style Sheets

JSON - JavaScript Object Notation

XML - Extensible Markup Language

Приложение Г

(справочное)

Библиографический список

1. Telegram или WhatsApp: какой мессенджер лучше? [Электронный ресурс]. – Режим доступа: https://perfluence.net/blog/article/telegram-ili-whatsapp.
2. Что такое No-code и Low-code [Электронный ресурс]. – Режим доступа: https://practicum.yandex.ru/blog/chto-takoe-no-code-i-low-code/.
3. Bcrypt - Wikipedia [Электронный ресурс]. – Режим доступа: https://en.wikipedia.org/wiki/Bcrypt.
4. JSON - Wikipedia [Электронный ресурс]. – Режим доступа: https://en.wikipedia.org/wiki/JSON.
5. XML - Wikipedia [Электронный ресурс]. – Режим доступа: https://en.wikipedia.org/wiki/XML.
6. The Go Programming Language [Электронный ресурс]. – Режим доступа: https://go.dev/.
7. PostgreSQL - Wikipedia [Электронный ресурс]. – Режим доступа: https://en.wikipedia.org/wiki/PostgreSQL.
8. Обзор протокола HTTP - HTTP | MDN - MDN Web Docs [Электронный ресурс]. – Режим доступа: https://developer.mozilla.org/ru/docs/Web/HTTP/Overview.
9. HTML | MDN - MDN Web Docs [Электронный ресурс]. – Режим доступа: https://developer.mozilla.org/ru/docs/Web/HTML.
10. CSS: каскадные таблицы стилей [Электронный ресурс]. – Режим доступа: https://developer.mozilla.org/ru/docs/Web/CSS.
11. JavaScript - Wikipedia [Электронный ресурс]. – Режим доступа: https://en.wikipedia.org/wiki/JavaScript.
12. TypeScript is JavaScript with syntax for types [Электронный ресурс] . – Режим доступа: https://www.typescriptlang.org/.