МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Вятский государственный университет»
Факультет автоматики и вычислительной техники
Кафедра электронных вычислительных машин

Допущено к защите
Руководитель проекта
_____ (Караваева О. В.)
«___» _____2023г.

Разработка прототипа обслуживающего сервиса конструктора Telegram ботов

Пояснительная записка курсового проекта по дисциплине
«Комплекс знаний бакалавра в области программного и аппаратного обеспечения вычислительной техники»
ТПЖА.090301.331 ПЗ

Разработал студент группы ИВТ-31 _____/Крючков И. С./

Руководитель _____/Караваева О. В./

Консультант _____/Кошкин О. В./

Проект защищен с оценкой «_____» _____
(оценка) (дата)

Члены комиссии _____/_____/
(подпись) (Ф.И.О.)

_____/_____/
(подпись) (Ф.И.О.)

_____/_____/
(подпись) (Ф.И.О.)

Киров 2023

Реферат

Крючков И. С. Разработка прототипа обслуживающего сервиса конструктора Telegram ботов. ТПЖА.090301.331 ПЗ: Курс. проект / ВятГУ, каф. ЭВМ; рук. Караваева О. В. - Киров, 2023. – ПЗ 96 с, 18 рис., 21 табл., 7 источников, 4 прил.

ОБСЛУЖИВАЮЩИЙ СЕРВИС, КОНСТРУКТОР, TELEGRAM БОТ API, GO, POSTGRESQL, REDIS, DOCKER, JSON.

Объект курсового проекта – информационный сервис, реализующий API для созданий и запуска Telegram ботов.

Цель курсового проекта – предоставить возможность создания Telegram ботов с помощью визуального конструктора.

Результат работы – разработанный прототип обслуживающего сервиса, предоставляющий пользователям функции по построению и запуску Telegram ботов.

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. Однако, для создания и управления Telegram ботами требуется определенный уровень технических знаний и навыков программирования, что может быть преградой для многих пользователей.

Обслуживающий сервис конструктора Telegram ботов является одной из частей платформы для создания и запуска Telegram ботов, которая позволит широкому кругу пользователей создавать и управлять Telegram ботами без необходимости программирования. Обслуживающий сервер отвечает за функции сервера для телеграм бота на общей структуре платформы. Структура платформы представлена на рисунке 1.

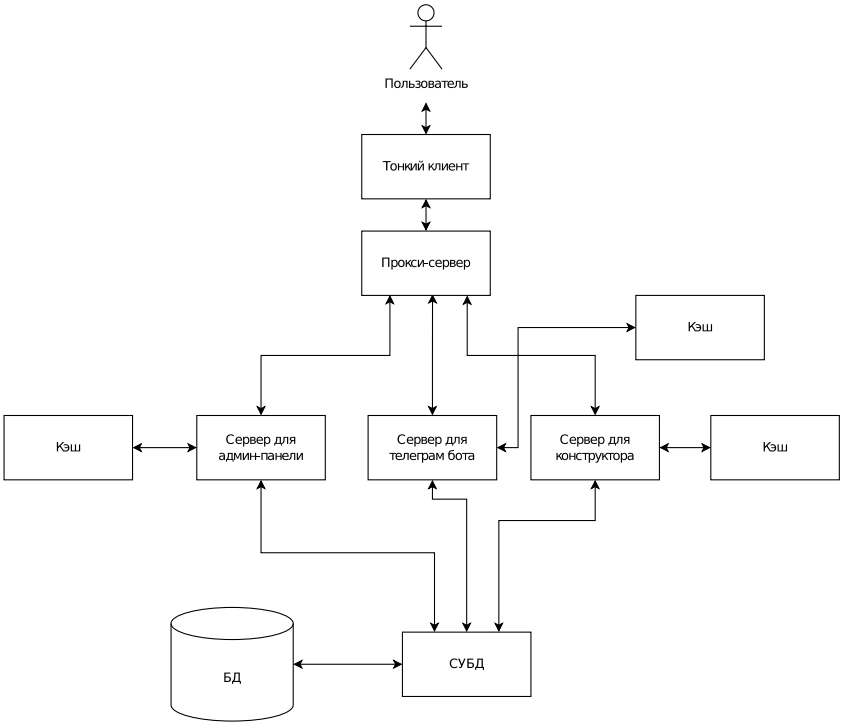

Рисунок 1 – Структура платформы

1 Анализ предметной области

В данном разделе проводится анализ предметной области, который позволит обосновать актуальность разработки проекта, приводятся ключевые требования и функциональность, которые должны быть реализованы в обслуживающем сервисе конструктора Telegram ботов

1.1 Актуальность темы

Боты в мессенджере Telegram становятся все более популярными и число их пользователей постоянно растет. Они помогают пользователям выполнять типичные рутинные действия в автоматизированном режиме, значительно упрощая им жизнь. Для владельцев же самих ботов они стали незаменимыми помощниками в работе.

Telegram-боты имеют множество плюсов, таких как:

– Круглосуточный доступ;

– Моментальный ответ на запрос пользователя;

– Удобство использования, интуитивно понятный интерфейс;

– Не требуют установки дополнительных программ, общение с ботом ведется через мессенджер;

– Широкий набор реализуемых функций

Telegram-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 NoCode инструмент, предназначенный 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.).

Первое предназначение – упрощение работы. Ведь не все обладают 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 Telegram-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.

Но у конструкторов есть некоторые ограничения, например, при их использовании нельзя выйти за рамки возможностей самого конструктора, а также при выходе нового функционала Telegram API, его реализация в конструкторе происходит с некоторой задержкой. Кроме того, боты, реализованные с помощью NoCode платформы могут быть менее производительные, чем их аналоги, написанные на языке программирования.

1.2 Постановка задачи

1.2.1 Основание для разработки

Обслуживающий сервис разрабатывается на основе задания на курсовое проектирование, полученного от заказчика в лице директора ООО «Синаптик», г. Киров.

1.2.2 Цель и задачи разработки

Целью разработки является реализация функционала для сокращения трудозатрат на создание и управление Telegram ботами.

Задачи:

– обеспечить функционал для создания Telegram ботов в среде визуального проектирования и их запуска;

– предусмотреть возможность задания собственного поведения бота при взаимодействии с пользователем;

1.2.3 Требования к функциональным характеристикам

Сервис должен обладать следующими функциональными возможностями.

Предоставлять REST API для управления ботом:

– создание нового бота в сервисе;

– получение списка ботов пользователя;

– установка и удаление Telegram Bot API токена бота;

– запуск и остановка работы бота.

Предоставлять REST API для редактирования структуры бота:

– получение структуры бота;

– добавление и удаление компонентов в структуру бота;

– редактирование компонентов;

– установка и удаление связей между компонентами в структуре;

– создание, редактирование и удаление команд в компонентах бота.

Описание компонентов, команд и связей между компонентами приводится в пункте 2.1.

Вывод

В данном разделе был проведен анализ предметной области и определены основные требования к разрабатываемому программному продукту.

Telegram боты являются функциональными инструментами для многих пользователей, однако для их разработки зачастую требуются навыки программирования, что усложняет их внедрение в бизнес-процессы компаний. Таким образом, проблема является актуальной, и разработка обслуживающего сервиса для платформы для создания ботов является целесообразной.

2 Разработка структуры сервиса

На данном этапе работы необходимо в соответствие с поставленными требованиями, разработать структуру сервиса, алгоритмы функционирования, механизмы взаимодействия компонентов информационной системы и структуру базы данных.

2.1 Разработка структуры сервиса

В рамках разработки структуры сервиса для обеспечения функционирования поставленных требований была разработана обобщенная структура, представляющая собой набор взаимосвязанный блоков, в которых будут реализованы алгоритмы функционирования обслуживающего сервиса. Структура представлена на рисунке 2.

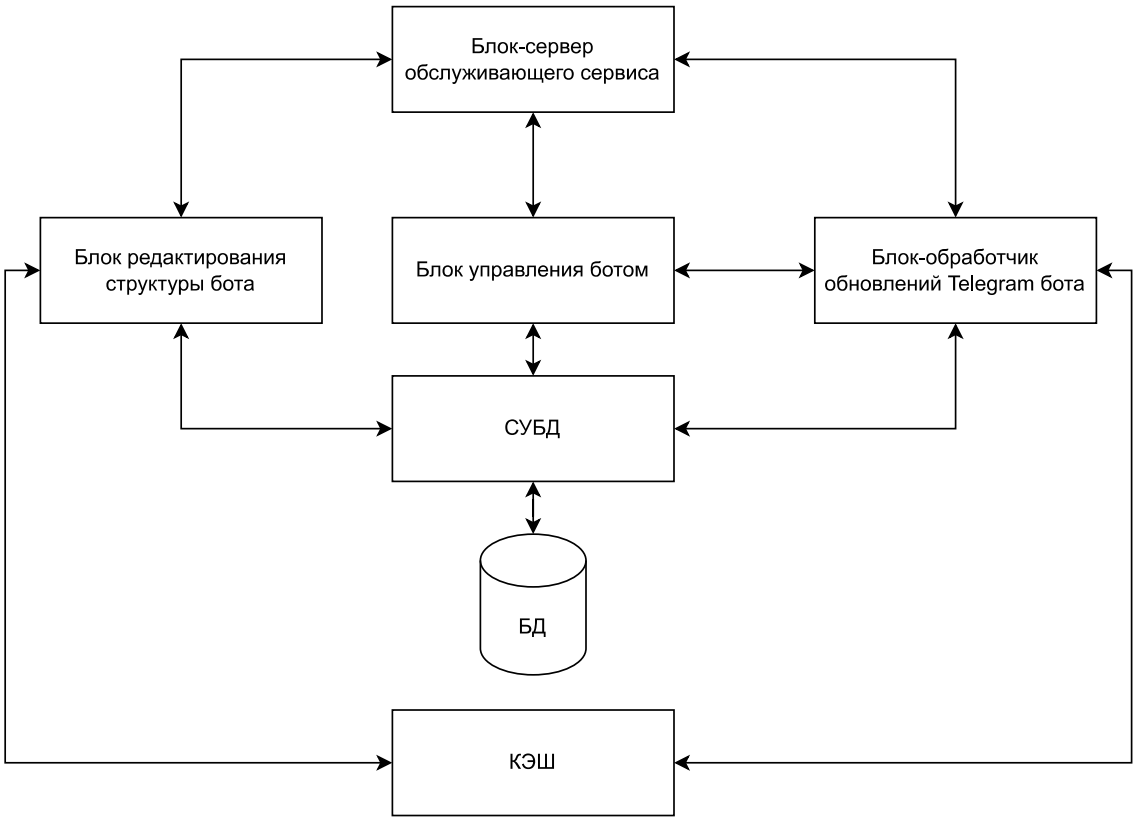

Рисунок 2 – Структура обслуживающего сервиса

Каждый из блоков выполняет следующие функции:

– блок управления ботом. Данный блок определяет алгоритмы запуска и остановки бота, установку и удаления API токена бота, а также его валидацию;

– блок редактирования структуры бота. Реализует функции добавления, удаления, редактирования компонентов в структуре бота. Добавление, изменение, удаление команд в компонентах, редактирование связей между компонентами структуры бота;

– блок-обработчик обновлений Telegram бота. Выполняет обработку новых запросов пользователей к каждому запущенному боту;

– блок-сервер обслуживающего сервиса. Предоставляет API интерфейс. Принимает запросы от клиента обслуживающего сервиса, выполняет в соответствие с вызванным методом его обработчик, возвращает результат выполнения клиенту сервиса.

Основные функции обслуживающего сервиса предназначены для работы с конкретным ботом.

Структура каждого бота состоит из следующих элементов:

– идентификатор – целое число, уникальное для каждого бота пользователя;

– идентификатор пользователя – целочисленное значение, id пользователя, которому принадлежит бот;

– название бота – строка, содержащая название бота в сервисе;

– API токен – строка содержащая API токен Telegram бота.

Реакция бота на входящие запросы пользователя определяется его структурой. Структура бота состоит из набора связанных компонентов, определяющих последовательность действий при работе пользователя с ботом.

Каждый компонент имеет в своей структуре следующие элементы:

– идентификатор – целое число, уникальное для каждого компонента в структуре бота;

– данные компонента – структура, содержащая в себе тип компонента и сами данные, специфичные для каждого типа компонента;

– структур клавиатуры – позиции размещения команд в Telegram боте;

– команды – список команд компонента;

– следующий шаг – целочисленное значение, соответствующее номеру компонента, в который будет выполнен переход из текущего компонента при работе бота;

– флаг начального компонента – булево значение, определяющее, является ли компонент начальным;

– позиция – объект с целочисленными координатами «x» и «y» задающими позицию компонента на поле визуального редактора структуры бота.

Одним из основных методов взаимодействия пользователя с Telegram ботом является отправка боту пользователем текстовых сообщений. Telegram Bot API позволяет при отправке сообщения ботом добавить специальную клавиатуру с предопределенными вариантами ответа, что является более интуитивно понятно для пользователя.

В структуре компонента имеется список команд, которые формируют клавиатуру.

Команда состоит из следующих элементов:

– идентификатор – целочисленное значение, уникальное для каждой команды в структуре бота;

– тип команды – строковое значение, определяющее функционал команды;

– данные команды – строка, содержащая данные, специфичные для каждого типа команды;

– идентификатор компонента – целое число, id компонента, которому принадлежит команда;

– следующий шаг – целочисленное значение, соответствующее номеру компонента, в который будет выполнен переход из текущего компонента при работе бота.

Так как разрабатывается прототип обслуживающего сервиса, список компонентов содержит минимально необходимый набор, для работоспособности бота. Список компонентов и их функционал содержится в таблице 1.

Таблица 1 – Список компонентов

Компонент	Функционал
Старт	Начальный компонент
Текстовое сообщение	Отправка текстового сообщения

При создании бота в обслуживающем сервисе, он содержим в своей структуре начальный компонент.

Для организации связанной структуры бота, каждый компонент может содержать ссылку на следующий компонент. Помимо этого, каждая команда в компоненте также может содержать в себе ссылку на компонент, который будет выполнен при отправке пользователем бота данной команды

2.2 Разработка основных алгоритмов функционирования

Для обеспечения взаимодействия с обслуживающим сервисом необходимо в соответствие с поставленной задачей предоставить набор API методов, реализующих основную логику работы сервиса.

2.2.1 Алгоритм взаимодействия с Telegram Bot API

Telegram Bot API поддерживает два взаимоисключающих способа получения обновлений (новых событий, сообщений и т.д) бота – «long polling» и «Webhook». Метод «long polling» рекомендуется использовать только на период разработки бота, в связи со спецификой его работы. При использовании данного метода, для получения информации о новых обновлениях от пользователей бота необходимо непрерывно с некоторой периодичностью опрашивать сервер Telegram на наличие обновлений бота выполняя соответствующий запрос к Telegram Bot API.

Webhook представляет собой механизм, который позволяет сервису получать входящие события от Telegram бота практически режиме реального времени. При использовании webhook, Telegram сохраняет URL-адрес (endpoint) сервиса, на который будут направляться все обновления, связанные с Telegram ботом.

Для того, чтобы включить режим работы webhook, необходимо выполнить запрос к методу «setWebhook» Telegram Bot API, указав токен бота, для которого будет установлен webhook и передав в теле запроса как минимум url адрес, на который будут поступать обновления бота.

После установки webhook, Telegram Bot API будет автоматически отправлять HTTP POST-запросы на указанный URL-адрес. Сервис получает эти запросы, извлекает необходимую информацию и обрабатывает ее соответствующим образом. Процесс взаимодействия пользователя с ботом показан на диаграмме последовательности – рисунок 3.

Webhook позволяет сервису моментально получать входящие обновления, без необходимости постоянного опроса Telegram Bot API на наличие новых событий. Это повышает эффективность и отзывчивость бота.

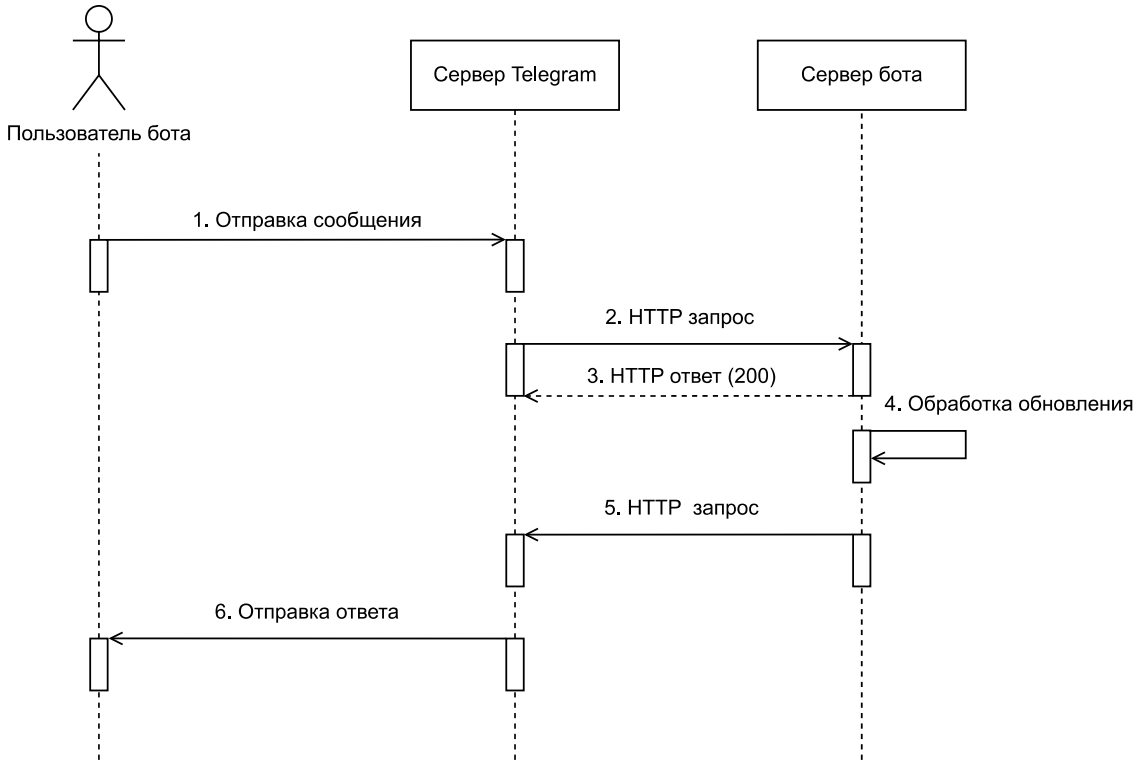

Рисунок 3 – Схема взаимодействия пользователя с ботом

2.2.2 Алгоритмы управления ботом

Для получения доступа к основным функциям сервиса, пользователю необходимо создать нового бота в обслуживающем сервисе. Добавление нового бота в обслуживающий сервис осуществляется вставкой записи в базу данных, с указанным названием бота. В результате создания бота, пользователю сервиса возвращается уникальный идентификатор данного бота в сервисе.

Параметры, передаваемые при вызове метода:

– название бота.

Ответ содержит:

– идентификатор бота;

– начальный компонент.

Доступ к Telegram Bot API происходит с помощью API токена, который выдается при создании бота в официальном боте Telegram @BotFather. Для сохранения токена бота в обслуживающем сервисе был разработан алгоритм установки токена для бота.

Алгоритм установки токена:

1) получить id бота, токен;
2) валидация токена:
   2.1) если токен не соответствует регулярному выражению, перейти к пункту 8;
   2.2) в противном случае перейти к пункту 3;
3) проверить существование бота:
   3.1) если бот не существует, перейти к пункту 8;
   3.2) иначе перейти к пункту 4;
4) проверить запущен ли бот в данный момент
   4.1) если бот запущен, перейти к пункту 8;
   4.2) в ином случае, перейти к пункту 5;
5) проверить существование токена в системе
   5.1) если указанный токен уже существует у какого-либо бота в системе, перейти к пункту 8;
   5.2) в противном случае перейти к пункту 6;
6) записать токен бота в БД;
7) вернуть положительный результат операции; завершить алгоритм;
8) вернуть ошибку.

Схема алгоритма установки токена представлена на рисунке 4.

Параметры, передаваемые при вызове метода:

– идентификатор бота.

Ответ содержит:

– результат выполнения метода.

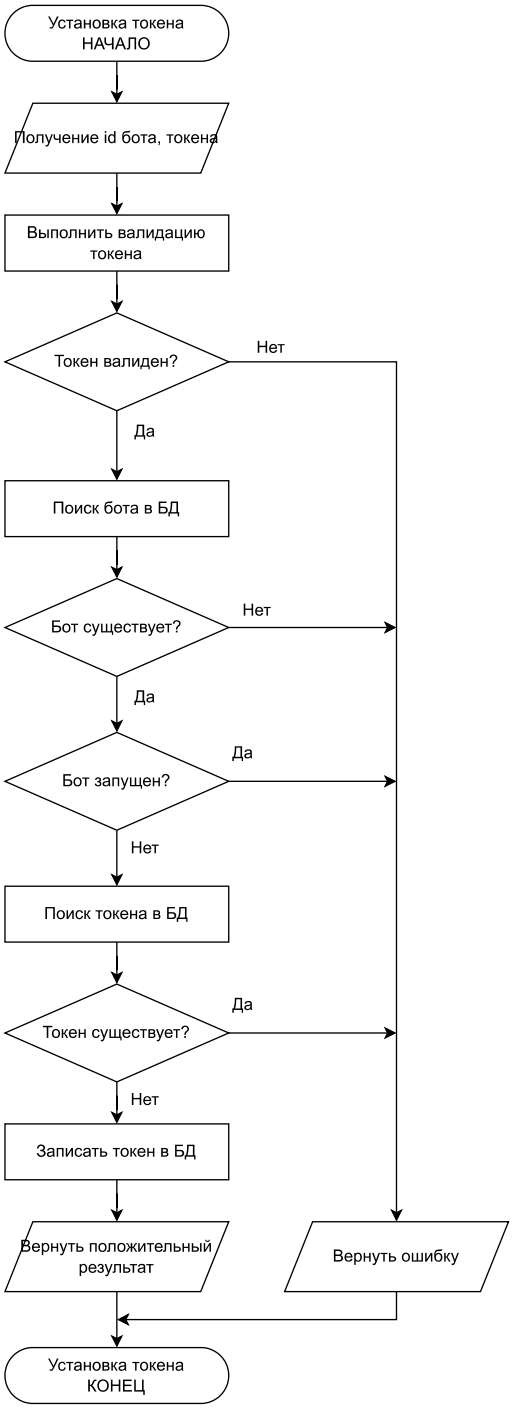

Рисунок 4 – Схема алгоритма «Установка токена»

Для удаления токена бота из сервиса необходимо проверить существование бота и его статус активности; токен может быть удален только в случае, если бот существует и не находится в запущенном состоянии. Если бот не существует или находится в запущенном состоянии, необходимо вернуть соответствующую ошибку.

Параметры, передаваемые при вызове метода:

– идентификатор бота.

Ответ содержит:

– результат выполнения метода.

Бот может реагировать на входящие запросы пользователя только в случае, если запущена функция обработчик для данного бота. Для запуска бота необходимо проверить существование бота в системе, статус активности бота; в случае, если бот уже запущен, необходимо вернуть соответствующий ответ; если бот не запущен, необходимо проверить наличие API токена и, если токен существует, установить webhook, запустить функцию обработчик входящих обновлений от пользователей бота; в случае, если бот не содержит токен, необходимо вернуть ошибку.

Схема алгоритма запуска бота представлена на рисунке 5.

Параметры, передаваемые при вызове метода:

– идентификатор бота.

Ответ содержит:

– результат выполнения метода.

Для остановки бота необходимо проверить существование бота и его статус активности; бот может быть остановлен только в случае, если он существует в системе и находится в запущенном состоянии. При остановке бота webhook удаляется. В случае, если бот не найден или уже остановлен, необходимо вернуть соответствующую ошибку.

Параметры, передаваемые при вызове метода:

– идентификатор бота.

Ответ содержит:

– результат выполнения метода.

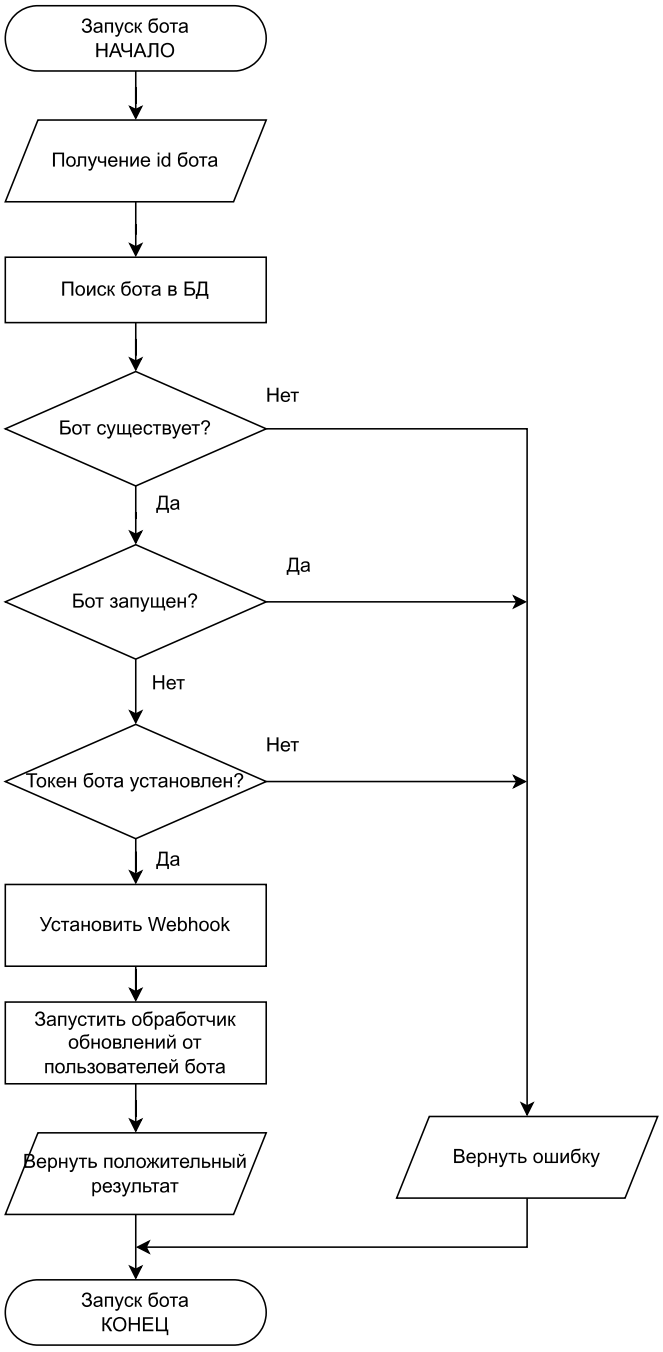

Рисунок 5 – Схема алгоритма «Запуск бота»

Получение списка всех ботов пользователя осуществляется с помощью соответствующего запроса в базу данных и возврата результата данного запроса.

Параметры, передаваемые при вызове метода: -

Ответ содержит:

– список ботов.

2.2.3 Алгоритмы редактирования структуры бота

Для добавления компонента в структуру бота, пользователю необходимо выполнить соответствующий запрос к обслуживающему сервису.

Алгоритм добавления компонента:

1) получить id бота и компонент;
2) валидация компонента:
   2.1) если тип компонента, данные, характерные для компонента данного типа и его позиция на поле редактора валидны, перейти к пункту 3;
   2.2) в противном случае, перейти к пункту 6;
3) проверить существование бота:
   3.1) если бот не существует, перейти к пункту 6;
   3.2) иначе перейти к пункту 4;
4) добавить компонент в БД;
5) вернуть положительный результат операции; завершить алгоритм;
6) вернуть ошибку.

Схема алгоритма добавления компонента в структуру бота представлена на рисунке 6.

Параметры, передаваемые при вызове метода:

- идентификатор бота;
- данные компонента;
- список команд компонента;
- позиция компонента.

Ответ содержит:

- идентификатор добавленного компонента.

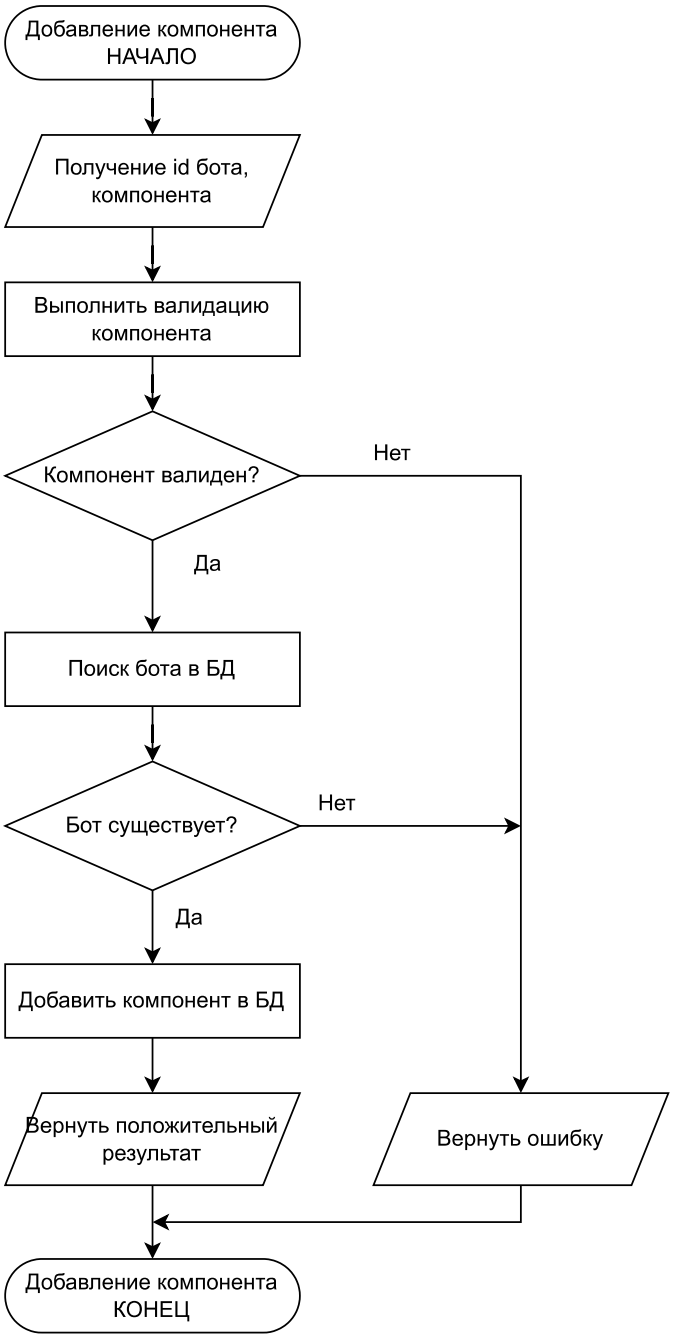

Рисунок 6 – Схема алгоритма «Добавление компонента»

Обновление существующего компонента может быть выполнено частично, указанием только необходимых полей, которые будут обновлены. При обновлении могут быть изменены данные компонента и его позиция.

Алгоритм обновления компонента:

1) получить новые значения полей компонента, id бота и id обновляемого компонента;

2) если получены данные или позиция компонента, то продолжить, иначе перейти к пункту 12;
3) если получены данные компонента, перейти к пункту 4, иначе к пункту 5;
4) валидация данных:
   4.1) если тип компонента и данные, характерные для компонента данного типа валидны, перейти к пункту 5;
   4.2) в противном случае, перейти к пункту 12;
5) если получена позиция компонента, перейти к пункту 6, в иначе к пункту 7;
6) валидация позиции:
   6.1) если позиция компонента валидна, перейти к пункту 7;
   6.2) в противном случае, перейти к пункту 12;
7) проверить существование бота:
   7.1) если бот не существует, перейти к пункту 12;
   7.2) иначе перейти к пункту 8;
8) проверить существование компонента:
   8.1) если компонент не существует, перейти к пункту 12;
   8.2) иначе перейти к пункту 9;
9) если получены данные компонента, обновить их в БД, иначе перейти к пункту 10;
10) если получена позиция компонента, обновить позицию в БД, иначе перейти к пункту 11;
11) вернуть положительный результат операции; завершить алгоритм;
12) вернуть ошибку.

Схема алгоритма обновления компонента представлена на рисунке 7.

Параметры, передаваемые при вызове метода:

– идентификатор бота;

– идентификатор компонента;

– данные компонента или позиция компонента.

Ответ содержит:

– результат выполнения метода.

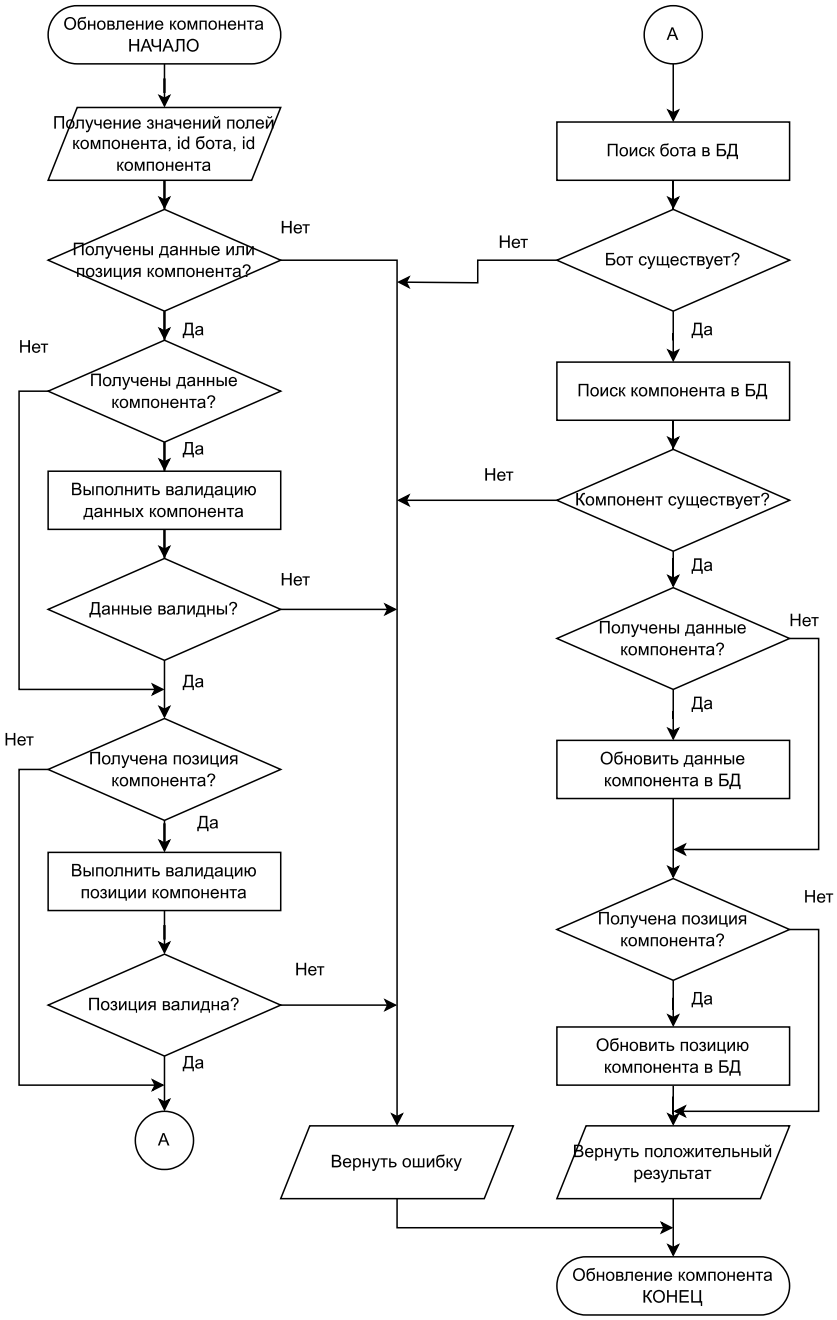

Рисунок 7 – Схема алгоритма «Обновление компонента»

Удаление существующего компонента из структуры бота может быть выполнено при условии, что данный компонента не является начальным, существует бот, из которого удаляется компонента и сам компонент существует в структуре бота. При удалении компонента также удаляются связанные с ним команды и происходит инвалидация кэша.

Схема алгоритма удаления компонента представлена на рисунке 8.

Параметры, передаваемые при вызове метода:

– идентификатор бота;

– идентификатор компонента.

Ответ содержит:

– результат выполнения метода.

Получение структуры бота осуществляется с помощью соответствующего запроса в базу данных и возврата результата данного запроса, при условии, что бот существует.

Параметры, передаваемые при вызове метода:

– идентификатор бота;

Ответ содержит:

– список компонентов.

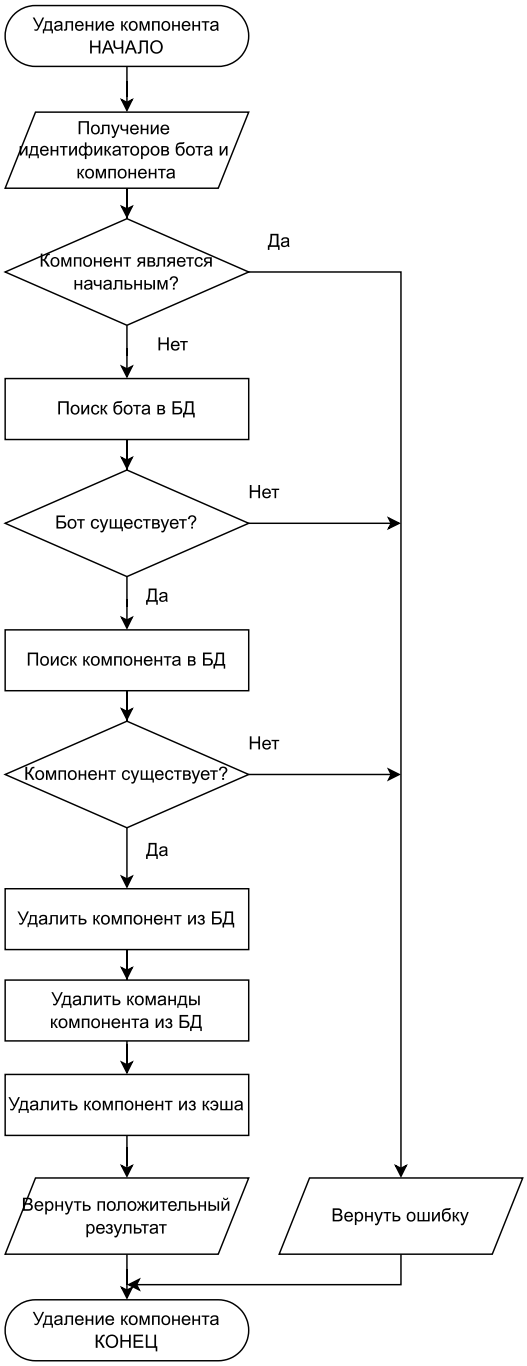

Рисунок 8 – Схема алгоритма «Удаление компонента»

В обслуживающем сервисе конструктора компоненты в структуре бота могут содержать команды, которые формируют клавиатуру с возможными вариантами ответа при отправке данных компонента ботом.

Алгоритм добавления команды:

1) получить команду, идентификатор бота и компонента, в который будет добавлена команда;

2) если команды добавляется к начальному компоненту, перейти к пункту 9, иначе к пункту 3;
3) валидация команды:
   3.1) если тип команды, данные, характерные для команды данного типа валидны, перейти к пункту 4;
   3.2) в противном случае, перейти к пункту 9;
4) проверить существование бота:
   4.1) если бот не существует, перейти к пункту 9;
   4.2) иначе перейти к пункту 5;
5) проверить существование компонента:
   5.1) если компонент не существует, перейти к пункту 9;
   5.2) иначе перейти к пункту 6;
6) добавить команду в БД;
7) удалить компонент, в который была добавлена команды из кэша;
8) вернуть положительный результат операции; завершить алгоритм;
9) вернуть ошибку.

Схема алгоритма добавления команды представлена на рисунке 9.

Параметры, передаваемые при вызове метода:

– идентификатор бота;
– идентификатор компонента;
– тип команды;
– данные команды.

Ответ содержит:

– идентификатор команды.

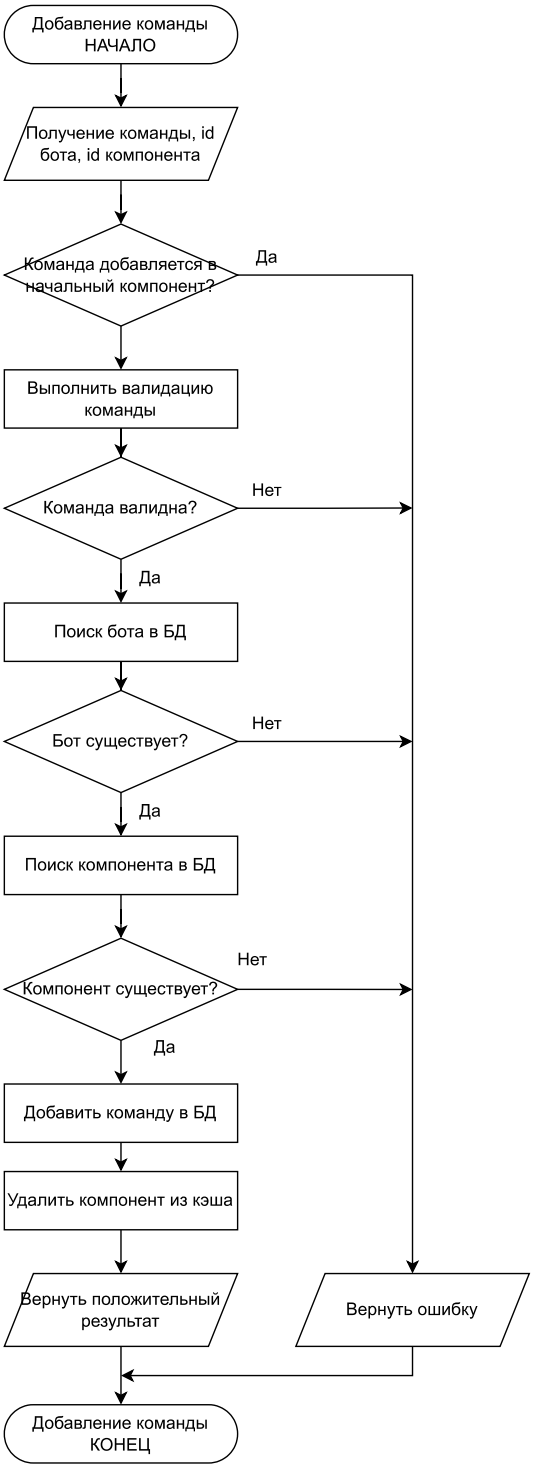

Рисунок 9 – Схема алгоритма «Добавление команды»

Алгоритм обновления команды:

1) получить значения типа и данных команды, идентификаторы бота, компонента, команды;
2) валидация команды:
   2.1) если тип команды, данные, характерные для команды данного типа валидны, перейти к пункту 3;
   2.2) в противном случае, перейти к пункту 9;
3) проверить существование бота:
   3.1) если бот не существует, перейти к пункту 9;
   3.2) иначе перейти к пункту 4;
4) проверить существование компонента:
   3.3) если компонент не существует, перейти к пункту 9;
   3.4) иначе перейти к пункту 5;
5) проверить существование команды:
   5.1) если команда не существует, перейти к пункту 9;
   5.2) иначе перейти к пункту 6;
6) обновить команду в БД;
7) удалить компонент из кэша;
8) вернуть положительный результат операции; завершить алгоритм;
9) вернуть ошибку.

Схема алгоритма обновления команды представлена на рисунке 10.

Параметры, передаваемые при вызове метода:

– идентификатор бота;
– идентификатор компонента;
– идентификатор команды;
– тип команды;
– данные команды.

Ответ содержит:

– результат выполнения метода.

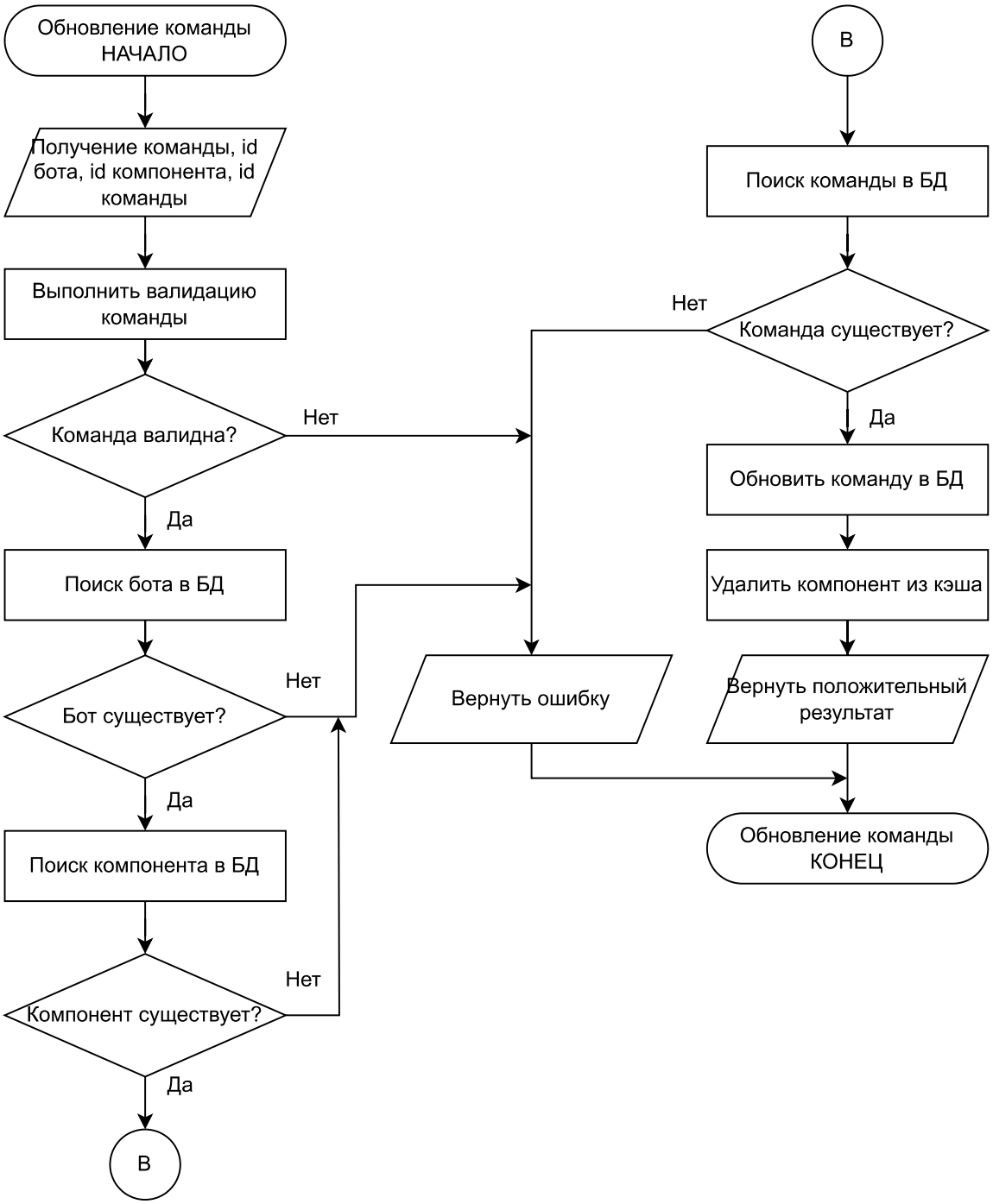

Рисунок 10 – Схема алгоритма «Обновление команды»

Удаление команды выполняется соответствующим запросов к БД, при этом должны быть выполнены условия существования бота, компонента, в котором находится команда, условие существования команды. После

удаления команды из БД, компонент, в котором она находилась удаляется из кэша.

Параметры, передаваемые при вызове метода:

– идентификатор бота;
– идентификатор компонента;
– идентификатор команды.

Ответ содержит:

– результат выполнения метода.

Для указания ссылки на следующий компонент необходимо вызвать соответствующий метод с указанием идентификаторов бота, начального компонента, которому устанавливается ссылка и следующего компонента.

Алгоритм установки ссылки на следующий компонент:

1) получить идентификаторы бота, начального компонента, следующего компонента;
2) если id начального компонента равен следующему, перейти к пункту 9, иначе к пункту 3;
3) проверить существование бота:
   3.1) если бот не существует, перейти к пункту 9;
   3.2) иначе перейти к пункту 4;
4) проверить существование начального компонента:
   3.3) если компонент не существует, перейти к пункту 9;
   3.4) иначе перейти к пункту 5;
5) проверить существование следующего компонента:
   3.5) если компонент не существует, перейти к пункту 9;
   3.6) иначе перейти к пункту 6;
6) установить в БД начальному компонента ссылку на следующий компонент;
7) удалить начальный компонент из кэша;

8) вернуть положительный результат операции; завершить алгоритм;

9) вернуть ошибку.

Схема алгоритма установки ссылки на следующий компонент представлена на рисунке 11.

Параметры, передаваемые при вызове метода:

– идентификатор бота;

– идентификатор компонента;

– идентификатор следующего компонента.

Ответ содержит:

– результат выполнения метода.

Удаление ссылки на следующий компонент происходит выполнением соответствующего запроса к БД, при этом необходимо проверить существование бота, и компонента, для которого удаляется ссылка. После удаления ссылки, компонент удаляется из кэша.

Параметры, передаваемые при вызове метода:

– идентификатор бота;

– идентификатор компонента.

Ответ содержит:

– результат выполнения метода.

Установка и удаление ссылки на следующий компонент для команды выполняются по схожим алгоритмам, за исключением необходимости в обоих алгоритмах выполнить дополнительную проверку на существование команды в компоненте.

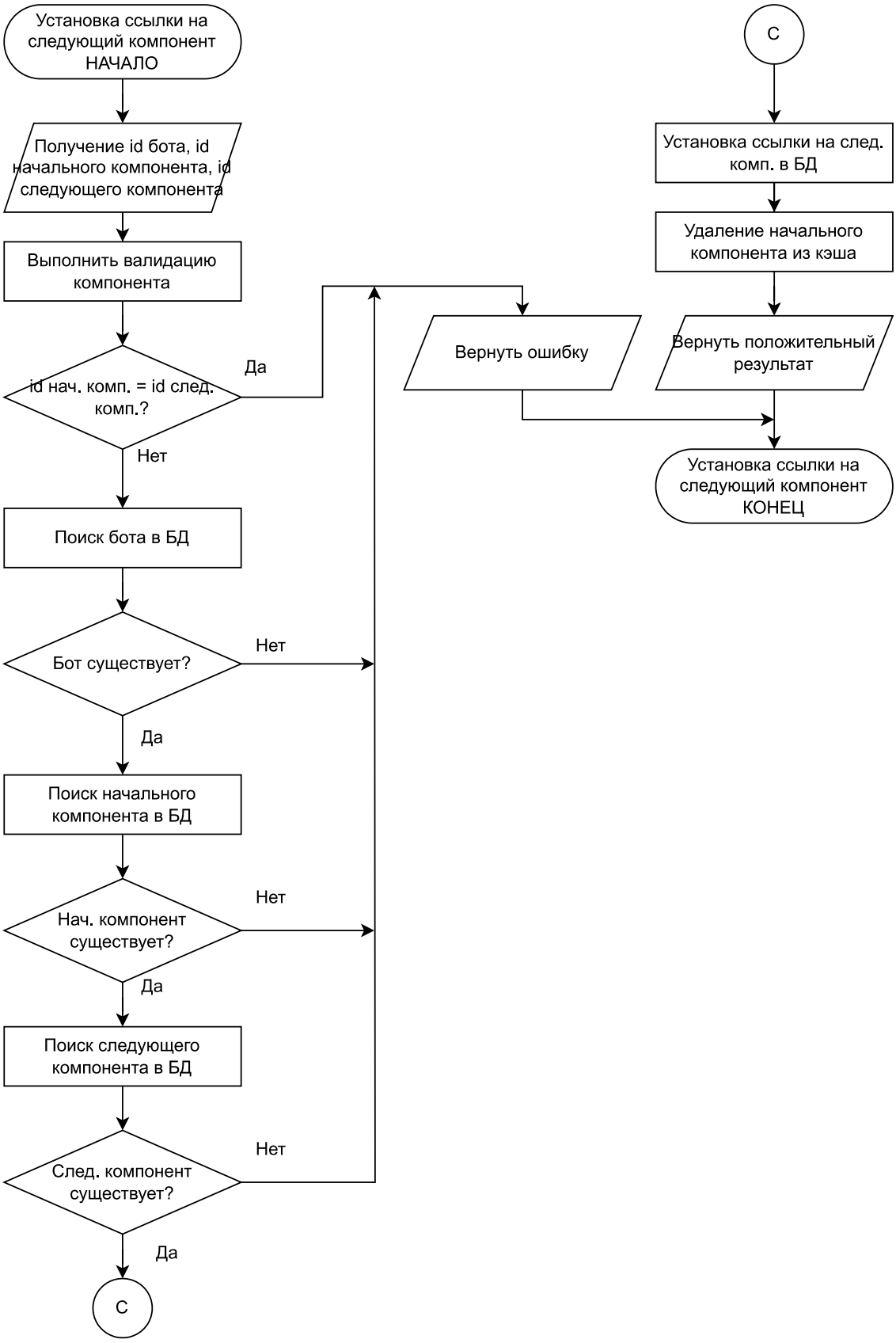

Рисунок 11 – Схема алгоритма «Установка ссылки на следующий компонент»

2.2.4 Алгоритмы обработки обновлений Telegram бота

Обработка входящих обновлений от пользователя Telegram бота является ключевым аспектом функционирования обслуживающего сервиса. Обслуживающий сервис реализует функции обработки команд Telegram бота (сообщение, начинающееся с «/», например «/start») и текстовых сообщений.

Обработка запросов пользователя к боту и формирование ответа происходит в соответствии с его структурой. Для каждого пользователя Telegram бота в обслуживающем сервисе хранится его шаг – идентификатор компонента из структуры бота, на котором он находится. Во время использования бота, шаг пользователя обновляется, в соответствии с шагом пользователя.

Для снижения нагрузки с базы данных и уменьшения времени обработки обновления структура бота и шаг пользователя хранится в кэше, так как эти данные используются при каждом обращении к боту.

Алгоритм обработки сообщений:

1) получить id пользователя и сообщение;
2) получить текущий шаг пользователя;
3) определить следующего компонента для выполнения:
   1.1) если компонент найден, перейти к пункту 4;
   1.2) иначе, перейти к пункту 6;
4) обновить шаг пользователя;
5) отправить ответ пользователю;
6) завершить алгоритм.

Схема алгоритма обработки сообщения представлена на рисунке 12.

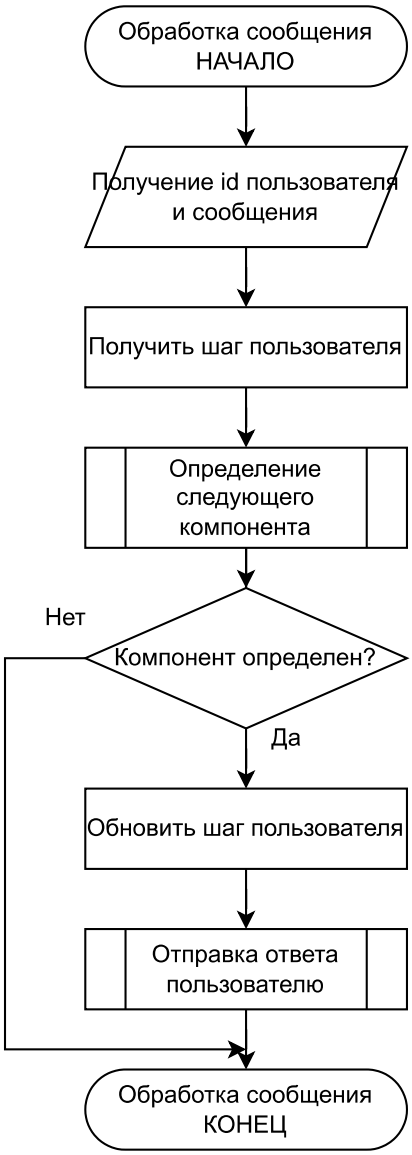

Рисунок 12 – Схема алгоритма «Обработка сообщения»

В связи с разработкой прототипа обслуживающего сервиса алгоритм обработки команд Telegram бота для обеспечения минимально необходимого функционала поддерживает только команду запуска бота «/start».

Алгоритм обработки команды Telegram бота:

1) получить id пользователя и команду;
2) если команда не равна «/start», перейти к пункту 6
3) определить следующего компонента для выполнения:
   3.1) если компонент найден, перейти к пункту 4;
   3.2) иначе, перейти к пункту 6;

4) обновить шага пользователя;

5) отправить ответ пользователю.

6) завершить алгоритм.

Схема алгоритма обработки команды Telegram бота представлена на рисунке 13.

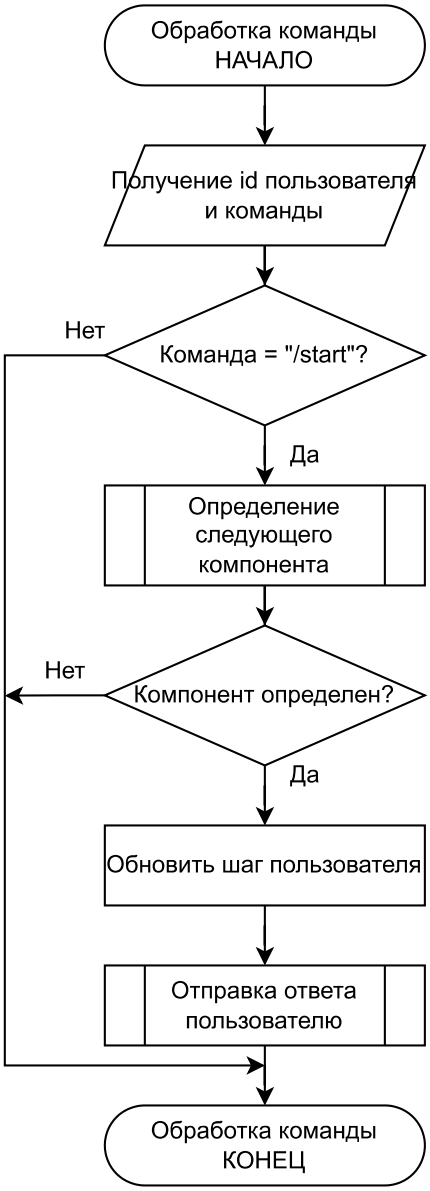

Рисунок 13 – Схема алгоритма «обработка команды»

Алгоритм определения следующего компонента можно условно разделить на два этапа. На первом этапе (пункты 1-7 алгоритма) происходит получение компонента, обработка стартового компонента, обнаружение зацикливания при выполнении переходов по структуре бота. На втором этапе (пункты 8-12 алгоритма) происходит определение следующего шага.

Алгоритм определения следующего компонента:

1) получить текущий шаг пользователя и сообщение;
2) сохранить шаг как исходный;
3) выполнить проверку на обнаружение цикла:
   3.1) если шаг уже проходили ранее, и он не равен исходному, перейти к пункту 11;
   3.2) если шаг уже проходили ранее, и он равен исходному, перейти к пункту 10;
   3.3) в противном случае, перейти к пункту 4
4) сохранить шаг, для проверки на зацикливание;
5) получить компонент для данного шага, сохранить как его исходный, если исходный не был установлен;
6) выполнить проверку на стартовый компонент:
   6.1) если компонент стартовый и не имеет ссылки на следующий компонент, перейти к пункту 11;
   6.2) если компонент стартовый и имеет ссылку на следующий компонент, установить текущий шаг, равным ссылке на след. компонент, установить флаг нахождения, перейти к пункту 3;
   6.3) в ином случае, перейти к пункту 7;
7) если установлен флаг нахождения, перейти к пункту 12, в ином случае установить данный флаг;

8) если у компонента установлена ссылка на следующий компонент, установить текущий шаг, равным данной ссылке, перейти к пункту 3, иначе продолжить;
9) выполнить поиск команды в компоненте в соответствии с текстом полученного сообщения:
   9.1) если команда найдена и у нее установлена ссылка на следующий компонент, установить текущий шаг, равным данной ссылке, перейти к пункту 3;
   9.2) в противном случае, перейти к пункту 10;
10) установить найденные шаг и компонент равные исходным, перейти к пункту 12;
11) вернуть ошибку, завершить алгоритм;
12) вернуть найденный шаг и компонент, завершить алгоритм.

Схема алгоритма определения следующего компонента представлена на рисунке 14.

Отправка ответа пользователю происходит путем обращения к определенному методу Telegram Bot API. Запрос формируется исходя из типа компонента в структуре бота (шаге, на котором находится пользователь) и данных, содержащихся в этом компоненте. При наличии в компоненте команд из них формируется клавиатура и добавляется к запросу. Схема алгоритма отправки ответа пользователю Telegram бота представлена на рисунке 15.

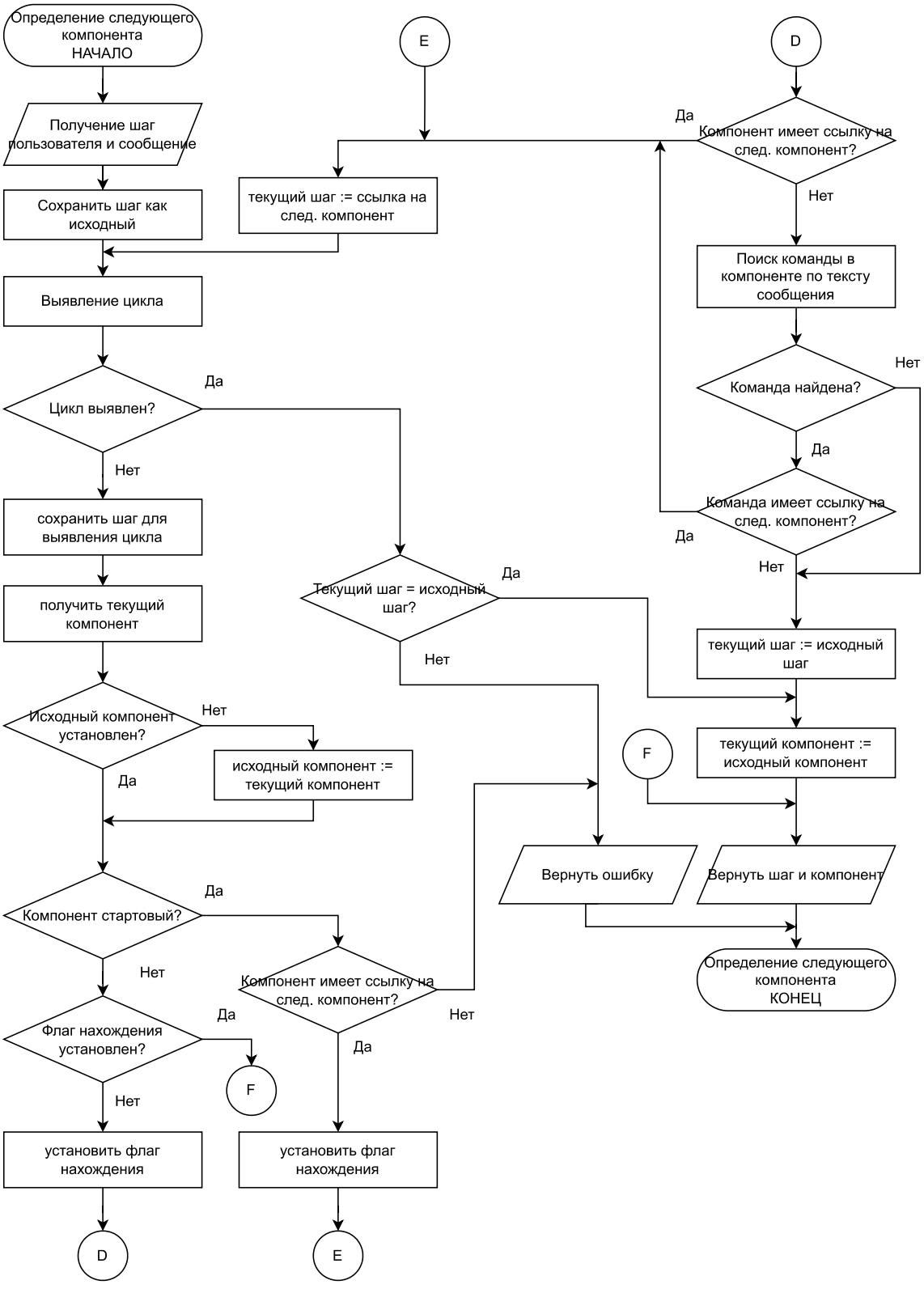

Рисунок 14 – Схема алгоритма «Определение следующего компонента»

Рисунок 15 – Схема алгоритма «Оправка ответа пользователю бота»

2.3 Разработка структуры базы данных

База данных — совокупность данных, организованных в соответствии с концептуальной структурой, описывающей характеристики этих данных и взаимоотношения между ними, которая поддерживает одну или более областей применения.

Рассмотрим два вида баз данных: реляционную и нереляционную базы данных.

Реляционная база данных (SQL) – это набор данных с предопределенными связями между ними. Эти данные организованы в виде набора таблиц (отношений), состоящих из столбцов и строк. В таблицах хранится информация об объектах, представленных в базе данных. Для взаимодействия с реляционными БД используется язык структурированных запросов - SQL.

Нереляционная база данных (NoSQL) – это база данных, в которой не используется табличная схема из строк и столбцов. В базах данных такого типа используется модель хранения, ориентированная на конкретные требования хранимых данных.

2.3.1 Концептуальная структура

Концептуальная модель является моделью наиболее высокого уровня абстракции, она создается без ориентации на какую-либо конкретную систему управления базами данных и модель данных. Она строится для описания и структуры базы данных. Концептуальная схема базы данных представлена на рисунке 16.

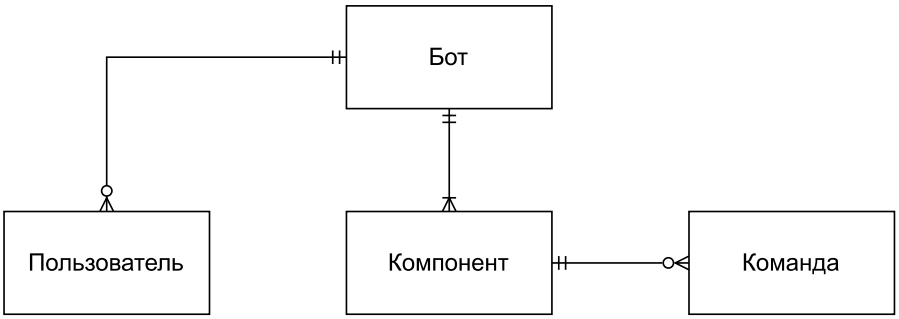

Рисунок 16 – Концептуальная схема базы данных

2.3.2 Логическая структура

Для проектирования логической схема базы данных используется реляционная модель, так как хранящиеся в ней данные представляют собой набор двумерных таблиц (отношений), в которых каждая строка уникальна и имеет один и тот же формат. Логическая схема базы данных представлена в виде ER диаграммы на рисунке 17.

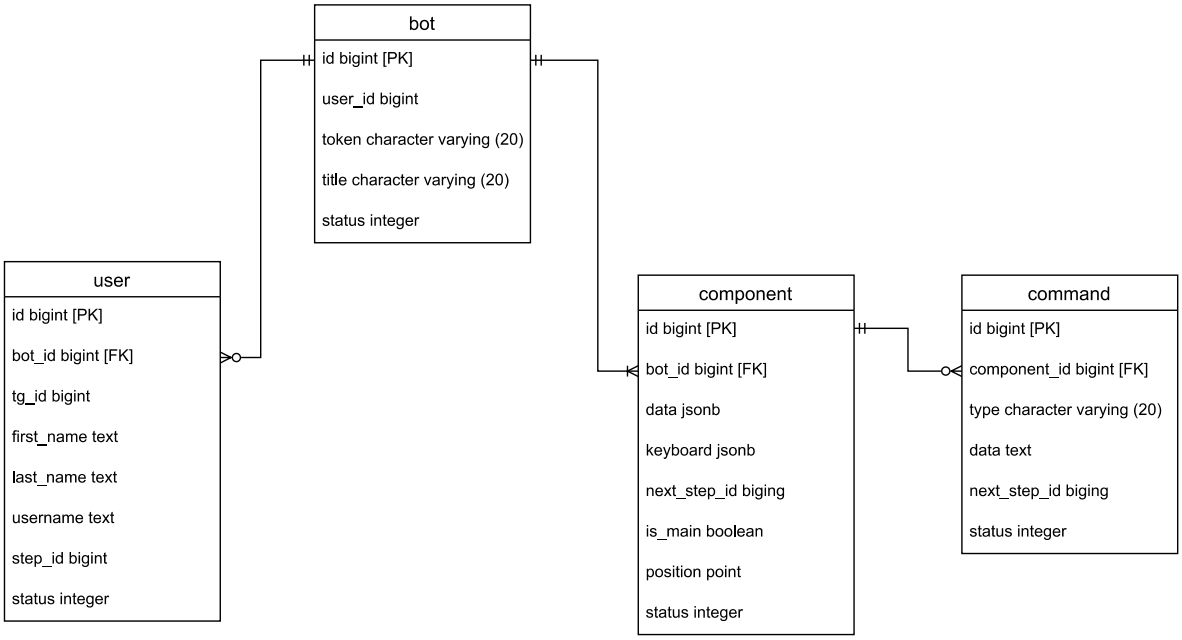

Рисунок 17 – ER-диаграмма структуры базы данных

Сущность «бот» является основополагающей: она содержит информацию о боте. Состоит из полей: уникальный идентификатор,

идентификатор пользователя, которому принадлежит бот, токен бота, название и статус.

Сущность «пользователь» служит для представления информации о пользователях бота. Содержит такие поля, как уникальный идентификатор, id бота, идентификатор пользователя в Telegram, имя, фамилию, username, номер шага в структуре бота и статус.

Сущность «компонент» содержит информацию о компоненте бота и состоит из следующих полей: уникальный идентификатор, идентификатор бота, данные компонента, структура клавиатуры, ссылка на следующий компонент, флаг, определяющий, является ли компонент начальным, позицию компонента и статус.

Сущность «команда» содержит уникальный идентификатор, id компонента, тип команды, данные команды, ссылку на следующий компонент, статус.

Вывод

В данном разделе была составлена структура обслуживающего сервиса конструктора Telegram ботов, разработаны алгоритмы управления ботом; редактирования структуры бота, позволяющие построить необходимую структуру, для определения последовательности действий при работе пользователя с ботом; алгоритмы обработки обновлений Telegram бота, обеспечивающие обработку поступающих событий Telegram бота и формирование ответа на них.

Были разработаны концептуальная и логическая схемы базы данных, содержащие сущности, необходимые для функционирования обслуживающего сервиса.

3 Программная реализация

В данном разделе представлена программная реализация обслуживающего сервиса конструктора Telegram ботом с использованием различных инструментов и технологий.

3.1 Выбор инструментов разработки

В качестве языка программирования для разработки был выбран Go. Go является компилируемым языком со строгой типизацией.

Основные преимущества языка:

– компилируемость – Go является компилируемым языков программирования, что обеспечивает высокую производительность за счет отсутствия необходимости в интерпретации;

– производительность – Go был разработан с упором на высокую производительность. Он эффективной системой управления памятью и оптимизированными механизмами работы с параллельными задачами. Это является важной особенностью при обработке большого количества одновременных запросов;

– встроенная поддержка параллелизма – в Go реализованы встроенные механизмы для эффективной работы с параллельными задачами. В частности, горутины (goroutines) и каналы позволяют разрабатывать программы, способные эффективно обрабатывать множество одновременных запросов.

Для хранения данных обслуживающего сервиса была выбрана свободная реляционная база данных PostgreSQL. PostgreSQL является одной из самых надежных и стабильных СУБД на рынке. База данных предоставляет возможность добавлять собственные функции, процедуры,

триггеры, индексы. В PostgreSQL имеется функционал схем. Схемы (schemas) представляют собой логические контейнеры, которые используются для организации и структурирования объектов базы данных, таких как таблицы, индексы, функции, представления и т.д. С помощью схем можно упростить управление и обеспечить четкую организацию данных в базе.

В качестве системы кэширования был выбран Redis. Redis является высокопроизводительной системой хранения данных типа ключ-значение в оперативной памяти. Используется как для баз данных, так и для реализации кэшей, брокеров сообщений. К преимуществам Redis относятся:

– высокая производительность – данные хранятся в оперативной памяти, что обеспечивает быстрый доступ к ним;

– поддержка различных структур данных - Redis поддерживает такие структуры данных, как строки, хэш-таблицы, множества, списки и другие;

– возможность сохранения данных на диск – Redis предоставляет возможность сохранять данные на диск, чтобы сохранить их при перезагрузке или сбое системы.

Docker был выбран для создания контейнерной среды разработки и развертывания обслуживающего сервиса. Docker – это открытая платформа для автоматизации развёртывания и запуска приложений в контейнерах. С его помощью можно упаковать приложение со всеми его зависимостями в контейнер, который может быть запущен на любой совместимой с Docker системе, независимо от окружения.

3.2 Формат передачи данных

Формат передачи данных в HTTP запросах, используемый в сервисе – 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.

Преимущество данного формата:

– простота использования – JSON представляет данные в виде текстового формата, которые легко читать, записывать как человеку, так и компьютеру;

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

– широкая поддержка – формат JSON поддерживается множеством языков программирования;

– универсальность – JSON поддерживает различные типы данных, включая строки, числа, массивы, объекты, булевы значения и т.д.

Результат каждого обращения к API методу представляет собой JSON объект, содержащий поля:

ok – булево значение, определяющее результата операции. Если ok = True, операция выполнена успешно, в ином случае, поле ok равняется false;

data – данные, полученные в результате выполнения метода;

error – поле, содержащее объект с информацией о коде и описании ошибки в случае её возникновения.

Поля data и error не являются обязательными и могут отсутствовать в зависимости от результата выполнения метода. Например, в случае

успешного выполнения метода, который не возвращает данные, в теле ответа API будет только поле ok.

Структура ответа API представлена на рисунке 18.

```
{
   "ok": "bool",
   "data":  "data",
   "error": {
      "code": "integer",
      "message": "string"
   }
}
```

Рисунок 18 – Структура ответа API

Работа с данными в формате JSON в Go выполняется с помощью пакета «goccy/go-json».

3.3 Структура объектов

В данном разделе содержится описание структуры основных объектов в обслуживающем сервисе конструктора Telegram ботов в формате JSON.

Описание структуры объектов представлено в таблице 2.

Описание структуры компонентов представлено в таблице 3.

Таблица 2 – Описание структуры объектов

Объект	Структура

Бот (bot)	{
	"id": "integer",
	"title": "string",
	}

Продолжение таблицы 2

Компонент (component)	{
	"id": "integer",
	"data": {
	"type": "string",
	"content": [
	"content"
	]
	},
	"keyboard": "keyboard",
	"commands": [
	"command"
	],
	"nextStepId": "integer",
	"isMain": "bool",
	"position": {
	"x": "integer",
	"y": "integer"
	}
	}

Команда (command)	{
	"id": "integer",
	"type": "string",
	"data": "string",
	"componentId": "integer",
	"nextStepId": "integer"
	}

Контент (content)	Содержит данные, специфичные для каждого компонента, см. таблицу 3.

Таблица 3 – Описание структуры компонентов

Компонент	Структура

Старт	{
	"type": "start",
	"content": []
	}

Текстовое сообщение	{
	"type": "start",
	"content": []
	}

3.4 Реализация функциональных блоков

Данный раздел содержит детали реализации разработанных функциональных блоков.

3.4.1 Блок управления ботом

В данном блоке реализованы алгоритмы управления ботом и представлено описание соответствующих API методов.

Создание нового бота.

Параметры HTTP запроса для создания нового бота представлены в таблице 4.

Таблица 4 – Параметры запроса «создание бота»

URL	/api/bots
HTTP Метод	POST

Параметры тела запроса	{
	"title": "string"
	}

Структура ответа	{
	"botId": "integer",
	"component": "component"
	}

Установка токена бота.

Параметры HTTP запроса для установки токена представлены в таблице 5.

Таблица 5 – Параметры запроса «установка токена бота»

URL	/api/bots/{botId}/token
HTTP Метод	POST
Параметры пути	botId: integer

Продолжение таблицы 5

Параметры тела запроса	{
	"token": "string"
	}

Структура ответа	-

Удаление токена бота.

Параметры HTTP запроса для удаления токена бота представлены в таблице 6.

Таблица 6 – Параметры запроса «удаление токена бота»

URL	/api/bots/{botId}/token
HTTP Метод	DELETE
Параметры пути	botId: integer
Параметры тела запроса	-
Структура ответа	-

Запуск бота.

Параметры HTTP запроса для запуска бота представлены в таблице 7.

Таблица 7 – Параметры запроса «запуск бота»

URL	/api/bots/{botId}/start
HTTP Метод	PATCH
Параметры пути	botId: integer
Параметры тела запроса	-
Структура ответа	-

Остановка бота.

Параметры HTTP запроса для остановки бота представлены в таблице 8.

Таблица 8 – Параметры запроса «запуск бота»

URL	/api/bots/{botId}/stop
HTTP Метод	PATCH
Параметры пути	botId: integer
Параметры тела запроса	-
Структура ответа	-

Получение списка ботов пользователя.

Параметры HTTP запроса для получения списка ботов представлены в таблице 9.

Таблица 9 – Параметры запроса «Получение списка ботов пользователя»

URL	/api/bots
HTTP Метод	GET
Параметры тела запроса	-
Структура ответа	["bot"]

3.4.2 Блок редактирования структуры бота

В данном блоке реализованы алгоритмы редактирования структуры бота и представлено описание соответствующих API методов.

Добавление компонента.

Параметры HTTP запроса для добавления компонента в структуру бота представлены в таблице 10.

Таблица 10 – Параметры запроса «добавление компонента»

URL	/api/bots/{botId}/components
HTTP Метод	POST
Параметры пути	botId: integer

Продолжение таблицы 10

Параметры тела запроса	{
	"data": {
	"type": "string",
	"content": [
	"content"
	]
	},
	"commands": [
	{
	"type": "string",
	"data": "string"
	},
	],
	"position": {
	"x": "integer",
	"y": "integer"
	}
	}

Структура ответа	{
	"id": "integer",
	}

Обновление компонента.

Параметры HTTP запроса для обновления компонента в структуре бота представлены в таблице 11.

Таблица 11 – Параметры запроса «обновление компонента»

URL	/api/bots/{botId}/components/{compId}
HTTP Метод	PATCH
Параметры пути	botId: integer compId: integer

Продолжение таблицы 11

Параметры тела запроса	{
	"data": {
	"type": "string",
	"content": [
	"content"
	]
	},
	"position": {
	"x": "integer",
	"y": "integer"
	}
	}

Структура ответа	-

Удаление компонента.

Параметры HTTP запроса для добавления компонента в структуру бота представлены в таблице 12.

Таблица 12 – Параметры запроса «удаление компонента»

URL	/api/bots/{botId}/components/{compId}
HTTP Метод	DELETE
Параметры пути	botId: integer compId: integer
Параметры тела запроса	-
Структура ответа	-

Получение структуры бота.

Параметры HTTP запроса для получения структуры бота представлены в таблице 13.

Таблица 13 – Параметры запроса «получение структуры бота»

URL	/api/bots/{botId}/components
HTTP Метод	GET
Параметры пути	botId: integer
Параметры тела запроса	-
Структура ответа	["component"]

Добавление команды.

Параметры HTTP запроса для добавления команды представлены в таблице 14.

Таблица 14 – Параметры запроса «добавление команды»

URL	/api/bots/{botId}/components/{compId}/commands
HTTP Метод	POST
Параметры пути	botId: integer compId: integer

Параметры тела запроса	{
	"type": "string",
	"data": "string"
	}

Структура ответа	{
	"id": "integer",
	}

Обновление команды.

Параметры HTTP запроса для обновления команды представлены в таблице 15.

Таблица 15 – Параметры запроса «обновление команды»

URL	/api/bots/{botId}/components/{compId}/commands/{commandId}
HTTP Метод	POST
Параметры пути	botId: integer compId: integer commandId: integer

Параметры тела запроса	{
	"type": "string",
	"data": "string"
	}

Структура ответа	-

Удаление команды.

Параметры HTTP запроса для удаления команды представлены в таблице 16.

Таблица 16 – Параметры запроса «удаление команды»

URL	/api/bots/{botId}/components/{compId}/commands/{commandId}
HTTP Метод	DELETE
Параметры пути	botId: integer compId: integer commandId: integer

Параметры тела запроса	{
	"type": "string",
	"data": "string"
	}

Структура ответа	-

Установка ссылки на следующий компонент.

Параметры HTTP запроса для установки ссылки на следующий компонент представлены в таблице 17.

Таблица 17 – Параметры запроса «установка ссылки на следующий компонент»

URL	/api/bots/{botId}/components/{compId}/next
HTTP Метод	POST
Параметры пути	botId: integer compId: integer

Параметры тела запроса	{
	"nextStepId": "integer"
	}

Структура ответа	-

Удаление ссылки на следующий компонент.

Параметры HTTP запроса для удаления ссылки на следующий компонент представлены в таблице 18.

Таблица 18 – Параметры запроса «удаление ссылки на следующий компонент»

URL	/api/bots/{botId}/components/{compId}/next
HTTP Метод	DELETE

Продолжение таблицы 18

Параметры пути	botId: integer compId: integer
Параметры тела запроса	-
Структура ответа	-

Установка команде ссылки на следующий компонент.

Параметры HTTP запроса для установки команде ссылки на следующий компонент представлены в таблице 19.

Таблица 19 – Параметры запроса «установка команде ссылки на следующий компонент»

URL	/api/bots/{botId}/components/{compId}/commands/{commandId}/next
HTTP Метод	POST
Параметры пути	botId: integer compId: integer commandId: integer

Параметры тела запроса	{
	"nextStepId": "integer"
	}

Структура ответа	-

Удаление из команды ссылки на следующий компонент.

Параметры HTTP запроса для удаления из команды ссылки на следующий компонент представлены в таблице 20.

Таблица 20 – Параметры запроса «удаление из команды ссылки на следующий компонент»

URL	/api/bots/{botId}/components/{compId}/commands/{commandId}/next
HTTP Метод	DELETE

Продолжение таблицы 20

Параметры пути	botId: integer compId: integer commandId: integer

Параметры тела запроса	{
	"nextStepId": "integer"
	}

Структура ответа	-

3.4.3 Блок-обработчик обновлений Telegram бота

В данном блоке реализованы алгоритмы обработки обновлений Telegram бота.

Обслуживающий сервис должен поддерживать параллельную обработку сообщений множества ботов. Получение обновлений от Telegram для бота происходит с помощью механизма webhook по уникальному адресу.

Шаблон URL, на который поступают обновления от Telegram бота: /webhook/botTOKEN, где TOKEN – токен бота. Использование токена в url обработчика является крайне нежелательным и в перспективе развития шаблон будет переработан.

Независимая обработка входящих обновлений обеспечивается запуском обработчика каждого бота в отдельной горутине. Горутины представляют собой легковесные потоки, которые работают независимо друг от друга. К их преимуществам относятся:

– легковесность – в сравнении с традиционными потоками, горутины не требуют большого объема памяти и могут быть созданы и уничтожены очень быстро. Таким образом могут быть созданы тысячи горутин без значительного влияния на производительность.

– каналы для коммуникации – обмен данными между горутинами происходит с помощью каналов (channels), которые обеспечивают

безопасную и эффективную синхронизацию и обмен данными между параллельными задачами;

– параллелизм – Go автоматически распределяет горутины на доступные процессоры, чтобы эффективно использовать вычислительные ресурсы многоядерных процессоров.

При получении каждого обновления от Telegram бота обработчик взаимодействует со структурой бота и шагом пользователя, которые хранятся в кэш-памяти. За счет этого уменьшается время обработки обновления. При необходимости установить новое значение шага пользователя он записывается в кэш-память и асинхронно обновляется в базе данных. Организация кэша приведена в разделе 3.5.

Пакет для работы с Telegram Bot API в Go - «mymmrac/telego».

3.4.4 Блок-сервер обслуживающего сервиса

Сервер предоставляет REST API для взаимодействия обслуживающего сервиса с клиентом. Сервер принимает запросы от клиента сервиса, обрабатывает их и возвращает результат клиенту.

HTTP сервер реализован с помощью пакета «valyala/fasthttp».

Роутер, используемый для регистрации обработчиков в данном сервере – «fasthttp/router»

3.5 Организация кэширования

В качестве системы кэширования используется Redis. Кэш используется для хранения шага пользователя и структуры бота.

Шаг пользователя хранится в виде хэш-таблицы с одним значение. В качестве ключа в таблице используется строка «step», в качестве значения шаг пользователя.

Сохранение в Redis выполняется по ключу формата: botBOT_ID:user:USER_TG_ID, где BOT_ID – идентификатор бота, USER_TG_ID – идентификатор пользователя Telegram.

Компоненты бота хранятся в виде хеш-таблицы, в качестве ключа в которой выступает идентификатор компонента, а значение – компонент в формате сериализованного JSON.

Компоненты бота сохраняются с ключом, формата: botBOT_ID:component, где BOT_ID – идентификатор бота.

В таблице 21 приводится преставление, данных хранимых в Redis.

Ключ	Значение
botBOT_ID:component	{ COMPONENT_ID: компонент },

botBOT_ID:user:USER_TG_ID	{
	"step": шаг пользователя
	}

Redis поддерживает возможность установки времени жизни для данных в памяти – TTL. Когда ключ с установленным TTL достигает времени своего завершения, Redis автоматически удаляет этот ключ и данные, связанные с ним из базы.

Удаление данных в обслуживающем сервисе из кэша происходит через час после последнего использования.

3.6 Организация контейнерной среды разработки

Разработка в контейнерной среде с использованием Docker и Docker compose. Docker compose используется для определения и управления

многоконтейнерным окружением, что упрощает развертывание и управление сервиса. Docker compose позволяет определить структуру и конфигурацию множества контейнеров, как единого приложения. В конфигурационном файле указываются параметры каждого контейнера, включая используемые Docker-образы, открытые порты, сетевые настройки, переменные окружения, примонтированные тома.

Функционирование обслуживающего сервиса обеспечивается набором следующих сервисов:

– bot – обслуживающий сервис Telegam;
– pgsql_bot – сервис PostgreSQL;
– redis_bot – сервис Redis.

Для взаимодействия между сервисами была организована виртуальная сеть Docker - bot_nw.

Текст конфигурационного файла Docker Compose приведен в приложении Б.

Вывод

На основе разработанной структуры сервиса была выполнена его программная реализация. Реализованы методы управления ботом, редактирования структуры бота и алгоритм обработчика обновлений бота. Основные фрагменты кода сервиса представлены в приложении А.

Выполнено организации системы кэширования частоиспользуемых данных.

Выполнена организация контейнерной среды разработки.

Заключение

В ходе выполнения курсового проекта по разработке прототипа обслуживающего сервиса конструктора Telegram ботов были проведены анализ предметной области, разработана структура сервиса, основные алгоритмы функционирования, разработана структура базы данных и осуществлена программная реализация.

Анализ предметной области позволил выявить основные требования и функциональность сервиса. Разработка основных алгоритмов функционирования позволила определить логику работы сервиса. В ходе программной реализации были определены инструменты разработки, формат и структура передаваемых данных.

В результате выполнения курсового проекта был разработан прототип обслуживающего сервиса конструктора Telegram ботов, предоставляющий пользователям API для создания и запуска Telegram ботов.

В качестве направления дальнейшего развития можно рассмотреть разбиение обслуживающего сервиса на микросервисы, добавление новых компонентов для построения структуры бота и другие улучшения.

Приложение А

(обязательное)

Фрагменты листинга кода

```
package handlers

import (
	"errors"
	"strconv"
	"unicode/utf8"

	"github.com/goccy/go-json"
	"go.uber.org/zap"

	e "github.com/botscubes/bot-service/internal/api/errors"
	"github.com/botscubes/bot-service/internal/bot"
	"github.com/botscubes/bot-service/internal/config"
	"github.com/botscubes/bot-service/internal/database/pgsql"
	rdb "github.com/botscubes/bot-service/internal/database/redis"
	"github.com/botscubes/bot-service/internal/model"
	resp "github.com/botscubes/bot-service/pkg/api_response"
	fh "github.com/valyala/fasthttp"
)

type newBotReq struct {
	Title *string `json:"title"`
}

type newBotRes struct {
	BotId     int64           `json:"botId"`
	Component *model.Component `json:"component"`
}

func NewBot(db *pgsql.Db, log *zap.SugaredLogger) reqHandler {
	return func(ctx *fh.RequestCtx) {
		var data newBotReq

		if err := json.Unmarshal(ctx.PostBody(), &data); err != nil {
			doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil, e.ErrInvalidRequest))
			return
		}

		title := data.Title
		userId, ok := ctx.UserValue("userId").(int64)
		if !ok {
			log.Error(ErrUserIDConvertation)
			doJsonRes(ctx, fh.StatusInternalServerError, resp.New(false, nil, e.ErrInternalServer))
			return
		}

		token := ""

		if title == nil || *title == "" {
			doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil, e.MissingParam("title")))
			return
		}
```

```
		if utf8.RuneCountInString(*title) > config.MaxTitleLen {
			doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil, e.ErrInvalidTitleLength))
			return
		}

		m := &model.Bot{
			UserId: userId,
			Token:  &token,
			Title:  title,
			Status: model.StatusBotActive,
		}

		botId, err := db.AddBot(m)
		if err != nil {
			log.Error(err)
			doJsonRes(ctx, fh.StatusInternalServerError, resp.New(false, nil, e.ErrInternalServer))
			return
		}

		if err := db.CreateBotSchema(botId); err != nil {
			log.Error(err)
			doJsonRes(ctx, fh.StatusInternalServerError, resp.New(false, nil, e.ErrInternalServer))
			return
		}

		dataType := "start"

		mc := &model.Component{
			Data: &model.Data{
				Type:    &dataType,
				Content: &[]*model.Content{},
			},
			Keyboard: &model.Keyboard{
				Buttons: [][]*int64{},
			},
			NextStepId: nil,
			IsMain:     true,
			Position: &model.Point{
				X: float64(config.StartComponentPosX), Y:
float64(config.StartComponentPosY),
				Valid: true,
			},
			Status: model.StatusComponentActive,
		}

		compId, err := db.AddComponent(botId, mc)
		if err != nil {
			log.Error(err)
			doJsonRes(ctx, fh.StatusInternalServerError, resp.New(false, nil, e.ErrInternalServer))
			return
		}

		mc.Id = compId
		mc.Commands = new(model.Commands)

		dataRes := &newBotRes{
			BotId:     botId,
			Component: mc,
		}
```

```
                doJsonRes(ctx, fh.StatusOK, resp.New(true, dataRes, nil))
        }
}

var ErrTgAuth401 = errors.New(`telego: health check: telego: getMe(): api: 401 "Unauthorized"`)

func StartBot(
        db *pgsql.Db,
        bs *bot.BotService,
        r *rdb.Rdb,
        log *zap.SugaredLogger,
) reqHandler {
        return func(ctx *fh.RequestCtx) {
                botId, err := strconv.ParseInt(ctx.UserValue("botId").(string), 10, 64)
                if err != nil {
                        doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil, e.ErrInvalidRequest))
                        return
                }

                userId, ok := ctx.UserValue("userId").(int64)
                if !ok {
                        log.Error(ErrUserIDConvertation)
                        doJsonRes(ctx, fh.StatusInternalServerError, resp.New(false, nil, e.ErrInternalServer))
                        return
                }

                existBot, err := db.CheckBotExist(userId, botId)
                if err != nil {
                        log.Error(err)
                        doJsonRes(ctx, fh.StatusInternalServerError, resp.New(false, nil, e.ErrInternalServer))
                        return
                }

                if !existBot {
                        doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil, e.ErrBotNotFound))
                        return
                }

                token, err := db.GetBotToken(userId, botId)
                if err != nil {
                        log.Error(err)
                        doJsonRes(ctx, fh.StatusInternalServerError, resp.New(false, nil, e.ErrInternalServer))
                        return
                }

                if token == nil || *token == "" {
                        doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil, e.ErrTokenNotFound))
                        return
                }

                if ok := bs.CheckBotExist(botId); !ok {
                        if err = bs.NewBot(token, botId, log, r, db); err != nil {
                                if err.Error() == ErrTgAuth401.Error() {
                                        doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil,
e.ErrInvalidToken))
                                        return
                                }
```

```
                                log.Error(err)
                                doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil, e.ErrNewBot))
                                return
                        }
                }

                // check bot already runnig
                isRunning, err := bs.BotIsRunnig(botId)
                if err != nil {
                        log.Error(err)
                        doJsonRes(ctx, fh.StatusInternalServerError, resp.New(false, nil, e.ErrInternalServer))
                        return
                }

                if isRunning {
                        doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil, e.ErrBotAlreadyRunning))
                        return
                }

                if err = bs.StartBot(botId); err != nil {
                        log.Error(err)
                        doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil, e.ErrStartBot))
                        return
                }

                doJsonRes(ctx, fh.StatusOK, resp.New(true, nil, nil))
        }
}

func StopBot(db *pgsql.Db, bs *bot.BotService, log *zap.SugaredLogger) reqHandler {
        return func(ctx *fh.RequestCtx) {
                botId, err := strconv.ParseInt(ctx.UserValue("botId").(string), 10, 64)
                if err != nil {
                        doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil, e.ErrInvalidRequest))
                        return
                }

                userId, ok := ctx.UserValue("userId").(int64)
                if !ok {
                        log.Error(ErrUserIDConvertation)
                        doJsonRes(ctx, fh.StatusInternalServerError, resp.New(false, nil, e.ErrInternalServer))
                        return
                }

                existBot, err := db.CheckBotExist(userId, botId)
                if err != nil {
                        log.Error(err)
                        doJsonRes(ctx, fh.StatusInternalServerError, resp.New(false, nil, e.ErrInternalServer))
                        return
                }

                if !existBot {
                        doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil, e.ErrBotNotFound))
                        return
                }

                token, err := db.GetBotToken(userId, botId)
                if err != nil {
                        log.Error(err)
```

```
                doJsonRes(ctx, fh.StatusInternalServerError, resp.New(false, nil, e.ErrInternalServer))
                return
            }

            if token == nil || *token == "" {
                doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil, e.ErrTokenNotFound))
                return
            }

            if ok := bs.CheckBotExist(botId); !ok {
                doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil, e.ErrBotAlreadyStopped))
                return
            }

            // check bot already stopped
            isRunning, err := bs.BotIsRunnig(botId)
            if err != nil {
                log.Error(err)
                doJsonRes(ctx, fh.StatusInternalServerError, resp.New(false, nil, e.ErrInternalServer))
                return
            }

            if !isRunning {
                doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil, e.ErrBotAlreadyStopped))
                return
            }

            if err := bs.StopBot(botId); err != nil {
                log.Error(err)
                doJsonRes(ctx, fh.StatusInternalServerError, resp.New(false, nil, e.ErrStopBot))
                return
            }

            doJsonRes(ctx, fh.StatusOK, resp.New(true, nil, nil))
        }
    }

    func GetBots(db *pgsql.Db, log *zap.SugaredLogger) reqHandler {
        return func(ctx *fh.RequestCtx) {
            userId, ok := ctx.UserValue("userId").(int64)
            if !ok {
                log.Error(ErrUserIDConvertation)
                doJsonRes(ctx, fh.StatusInternalServerError, resp.New(false, nil, e.ErrInternalServer))
                return
            }

            bots, err := db.UserBots(userId)
            if err != nil {
                log.Error(err)
                doJsonRes(ctx, fh.StatusInternalServerError, resp.New(false, nil, e.ErrInternalServer))
                return
            }

            doJsonRes(ctx, fh.StatusOK, resp.New(true, bots, nil))
        }
    }

    func WipeBot(db *pgsql.Db, r *rdb.Rdb, bs *bot.BotService, log *zap.SugaredLogger) reqHandler {
        return func(ctx *fh.RequestCtx) {
```

```
		botId, err := strconv.ParseInt(ctx.UserValue("botId").(string), 10, 64)
		if err != nil {
			doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil, e.ErrInvalidRequest))
			return
		}

		userId, ok := ctx.UserValue("userId").(int64)
		if !ok {
			log.Error(ErrUserIDConvertation)
			doJsonRes(ctx, fh.StatusInternalServerError, resp.New(false, nil, e.ErrInternalServer))
			return
		}

		// check bot exists
		existBot, err := db.CheckBotExist(userId, botId)
		if err != nil {
			log.Error(err)
			doJsonRes(ctx, fh.StatusInternalServerError, resp.New(false, nil, e.ErrInternalServer))
			return
		}

		if !existBot {
			doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil, e.ErrBotNotFound))
			return
		}

		// stop bot worker
		if ok := bs.CheckBotExist(botId); ok {
			// check bot already stopped
			isRunning, err := bs.BotIsRunnig(botId)
			if err != nil {
				log.Error(err)
				doJsonRes(ctx, fh.StatusInternalServerError, resp.New(false, nil,
e.ErrInternalServer))
				return
			}

			if isRunning {
				if err := bs.StopBot(botId); err != nil {
					log.Error(err)
					doJsonRes(ctx, fh.StatusInternalServerError, resp.New(false, nil,
e.ErrStopBot))
					return
				}
			}
		}

		// remove components
		err = db.DelAllComponents(botId)
		if err != nil {
			log.Error(err)
			doJsonRes(ctx, fh.StatusInternalServerError, resp.New(false, nil, e.ErrInternalServer))
			return
		}

		// remove commands
		err = db.DelAllCommands(botId)
		if err != nil {
			log.Error(err)
```

```
                        doJsonRes(ctx, fh.StatusInternalServerError, resp.New(false, nil, e.ErrInternalServer))
                        return
                }

                // remove next step from main component
                if err = db.DelNextStepComponent(botId, config.MainComponentId); err != nil {
                        log.Error(err)
                        doJsonRes(ctx, fh.StatusInternalServerError, resp.New(false, nil, e.ErrInternalServer))
                        return
                }

                // remove token
                token := ""
                if err = db.SetBotToken(userId, botId, &token); err != nil {
                        log.Error(err)
                        doJsonRes(ctx, fh.StatusInternalServerError, resp.New(false, nil, e.ErrInternalServer))
                        return
                }

                // Invalidate bot cache
                r.DelBotData(botId)

                doJsonRes(ctx, fh.StatusOK, resp.New(true, nil, nil))
        }
}
package handlers

import (
        "strconv"

        "github.com/goccy/go-json"
        fh "github.com/valyala/fasthttp"
        "go.uber.org/zap"

        e "github.com/botscubes/bot-service/internal/api/errors"
        "github.com/botscubes/bot-service/internal/config"
        "github.com/botscubes/bot-service/internal/database/pgsql"
        rdb "github.com/botscubes/bot-service/internal/database/redis"
        "github.com/botscubes/bot-service/internal/model"
        resp "github.com/botscubes/bot-service/pkg/api_response"
)

type addComponentReq struct {
        Data     *model.Data     `json:"data"`
        Commands *model.Commands `json:"commands"`
        Position *model.Point    `json:"position"`
}

type addComponentRes struct {
        Id int64 `json:"id"`
}

func AddComponent(db *pgsql.Db, log *zap.SugaredLogger) reqHandler {
        return func(ctx *fh.RequestCtx) {
                botId, err := strconv.ParseInt(ctx.UserValue("botId").(string), 10, 64)
                if err != nil {
                        doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil, e.ErrInvalidRequest))
                        return
                }
```

```
		var reqData addComponentReq
		if err = json.Unmarshal(ctx.PostBody(), &reqData); err != nil {
			doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil, e.ErrInvalidRequest))
			return
		}

		// TODO: check fields limits:
		// eg. data.commands._.data max size, check commands max count
		if err := model.ValidateComponent(reqData.Data, reqData.Commands, reqData.Position); err !=
nil {
			doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil, err))
			return
		}

		userId, ok := ctx.UserValue("userId").(int64)
		if !ok {
			log.Error(ErrUserIDConvertation)
			doJsonRes(ctx, fh.StatusInternalServerError, resp.New(false, nil, e.ErrInternalServer))
			return
		}

		// check bot exists
		existBot, err := db.CheckBotExist(userId, botId)
		if err != nil {
			log.Error(err)
			doJsonRes(ctx, fh.StatusInternalServerError, resp.New(false, nil, e.ErrInternalServer))
			return
		}

		if !existBot {
			doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil, e.ErrBotNotFound))
			return
		}

		component := &model.Component{
			Data: reqData.Data,
			Keyboard: &model.Keyboard{
				Buttons: [][]*int64{},
			},
			NextStepId: nil,
			IsMain:     false,
			Position:   reqData.Position,
			Status:     model.StatusComponentActive,
		}

		compId, err := db.AddComponent(botId, component)
		if err != nil {
			log.Error(err)
			doJsonRes(ctx, fh.StatusInternalServerError, resp.New(false, nil, e.ErrInternalServer))
			return
		}

		for _, v := range *reqData.Commands {
			mc := &model.Command{
				Type:        v.Type,
				Data:        v.Data,
				ComponentId: &compId,
				NextStepId:  nil,
```

```
                                Status:     model.StatusCommandActive,
                        }

                        _, err := db.AddCommand(botId, mc)
                        if err != nil {
                                log.Error(err)
                                doJsonRes(ctx, fh.StatusInternalServerError, resp.New(false, nil,
e.ErrInternalServer))
                                return
                        }
                }

                dataRes := &addComponentRes{
                        Id: compId,
                }

                doJsonRes(ctx, fh.StatusOK, resp.New(true, dataRes, nil))
        }
}

type setNextStepComponentReq struct {
        NextStepId *int64 `json:"nextStepId"`
}

func SetNextStepComponent(db *pgsql.Db, r *rdb.Rdb, log *zap.SugaredLogger) reqHandler {
        return func(ctx *fh.RequestCtx) {
                botId, err := strconv.ParseInt(ctx.UserValue("botId").(string), 10, 64)
                if err != nil {
                        doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil, e.ErrInvalidRequest))
                        return
                }

                compId, err := strconv.ParseInt(ctx.UserValue("compId").(string), 10, 64)
                if err != nil {
                        doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil, e.ErrInvalidRequest))
                        return
                }

                var reqData setNextStepComponentReq
                if err = json.Unmarshal(ctx.PostBody(), &reqData); err != nil {
                        doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil, e.ErrInvalidRequest))
                        return
                }

                if reqData.NextStepId == nil {
                        doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil,
e.MissingParam("nextStepId")))
                        return
                }

                nextComponentId := reqData.NextStepId

                if *nextComponentId == compId {
                        doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil, e.InvalidParam("nextStepId")))
                        return
                }

                userId, ok := ctx.UserValue("userId").(int64)
                if !ok {
```

```
                log.Error(ErrUserIDConvertation)
                doJsonRes(ctx, fh.StatusInternalServerError, resp.New(false, nil, e.ErrInternalServer))
                return
            }

            // check bot exists
            existBot, err := db.CheckBotExist(userId, botId)
            if err != nil {
                log.Error(err)
                doJsonRes(ctx, fh.StatusInternalServerError, resp.New(false, nil, e.ErrInternalServer))
                return
            }

            if !existBot {
                doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil, e.ErrBotNotFound))
                return
            }

            // check bot component exists
            existInitialComp, err := db.CheckComponentExist(botId, compId)
            if err != nil {
                log.Error(err)
                doJsonRes(ctx, fh.StatusInternalServerError, resp.New(false, nil, e.ErrInternalServer))
                return
            }

            if !existInitialComp {
                doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil, e.ErrComponentNotFound))
                return
            }

            // check bot next component exists
            existNextComp, err := db.CheckComponentExist(botId, *nextComponentId)
            if err != nil {
                log.Error(err)
                doJsonRes(ctx, fh.StatusInternalServerError, resp.New(false, nil, e.ErrInternalServer))
                return
            }

            if !existNextComp {
                doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil,
e.ErrNextComponentNotFound))
                return
            }

            if err = db.SetNextStepComponent(botId, compId, *nextComponentId); err != nil {
                log.Error(err)
                doJsonRes(ctx, fh.StatusInternalServerError, resp.New(false, nil, e.ErrInternalServer))
                return
            }

            // Invalidate component cache
            if err = r.DelComponent(botId, compId); err != nil {
                log.Error(err)
            }

            doJsonRes(ctx, fh.StatusOK, resp.New(true, nil, nil))
        }
    }
```

```
func GetBotComponents(db *pgsql.Db, log *zap.SugaredLogger) reqHandler {
	return func(ctx *fh.RequestCtx) {
		botId, err := strconv.ParseInt(ctx.UserValue("botId").(string), 10, 64)
		if err != nil {
			doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil, e.ErrInvalidRequest))
			return
		}

		userId, ok := ctx.UserValue("userId").(int64)
		if !ok {
			log.Error(ErrUserIDConvertation)
			doJsonRes(ctx, fh.StatusInternalServerError, resp.New(false, nil, e.ErrInternalServer))
			return
		}

		// check bot exists
		existBot, err := db.CheckBotExist(userId, botId)
		if err != nil {
			log.Error(err)
			doJsonRes(ctx, fh.StatusInternalServerError, resp.New(false, nil, e.ErrInternalServer))
			return
		}

		if !existBot {
			doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil, e.ErrBotNotFound))
			return
		}

		components, err := db.ComponentsForEd(botId)
		if err != nil {
			log.Error(err)
			doJsonRes(ctx, fh.StatusInternalServerError, resp.New(false, nil, e.ErrInternalServer))
			return
		}

		doJsonRes(ctx, fh.StatusOK, resp.New(true, components, nil))
	}
}

func DelComponent(db *pgsql.Db, r *rdb.Rdb, log *zap.SugaredLogger) reqHandler {
	return func(ctx *fh.RequestCtx) {
		botId, err := strconv.ParseInt(ctx.UserValue("botId").(string), 10, 64)
		if err != nil {
			doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil, e.ErrInvalidRequest))
			return
		}

		compId, err := strconv.ParseInt(ctx.UserValue("compId").(string), 10, 64)
		if err != nil {
			doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil, e.ErrInvalidRequest))
			return
		}

		// check component is main
		if compId == config.MainComponentId {
			doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil, e.ErrMainComponent))
			return
		}
```

```
userId, ok := ctx.UserValue("userId").(int64)
if !ok {
	log.Error(ErrUserIDConvertation)
	doJsonRes(ctx, fh.StatusInternalServerError, resp.New(false, nil, e.ErrInternalServer))
	return
}

// check bot exists
existBot, err := db.CheckBotExist(userId, botId)
if err != nil {
	log.Error(err)
	doJsonRes(ctx, fh.StatusInternalServerError, resp.New(false, nil, e.ErrInternalServer))
	return
}

if !existBot {
	doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil, e.ErrBotNotFound))
	return
}

// check bot component exists
existComp, err := db.CheckComponentExist(botId, compId)
if err != nil {
	log.Error(err)
	doJsonRes(ctx, fh.StatusInternalServerError, resp.New(false, nil, e.ErrInternalServer))
	return
}

if !existComp {
	doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil, e.ErrComponentNotFound))
	return
}

if err = db.DelComponent(botId, compId); err != nil {
	log.Error(err)
	doJsonRes(ctx, fh.StatusInternalServerError, resp.New(false, nil, e.ErrInternalServer))
	return
}

if err = db.DelCommandsByCompId(botId, compId); err != nil {
	log.Error(err)
	doJsonRes(ctx, fh.StatusInternalServerError, resp.New(false, nil, e.ErrInternalServer))
	return
}

// remove component next steps, that reference these component
if err = db.DelNextStepComponentByNS(botId, compId); err != nil {
	log.Error(err)
	doJsonRes(ctx, fh.StatusInternalServerError, resp.New(false, nil, e.ErrInternalServer))
	return
}

// Invalidate component cache
if err = r.DelComponent(botId, compId); err != nil {
	log.Error(err)
}

doJsonRes(ctx, fh.StatusOK, resp.New(true, nil, nil))
```

```
	}
}


type addCommandReq struct {
	Type *string `json:"type"`
	Data *string `json:"data"`
}

type addCommandRes struct {
	Id int64 `json:"id"`
}

func AddCommand(db *pgsql.Db, r *rdb.Rdb, log *zap.SugaredLogger) reqHandler {
	return func(ctx *fh.RequestCtx) {
		botId, err := strconv.ParseInt(ctx.UserValue("botId").(string), 10, 64)
		if err != nil {
			doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil, e.ErrInvalidRequest))
			return
		}

		compId, err := strconv.ParseInt(ctx.UserValue("compId").(string), 10, 64)
		if err != nil {
			doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil, e.ErrInvalidRequest))
			return
		}

		// check component is main
		if compId == config.MainComponentId {
			doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil, e.ErrMainComponent))
			return
		}

		var reqData addCommandReq
		if err = json.Unmarshal(ctx.PostBody(), &reqData); err != nil {
			doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil, e.ErrInvalidRequest))
			return
		}

		if err := model.ValidateCommand(reqData.Type, reqData.Data); err != nil {
			doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil, err))
			return
		}

		userId, ok := ctx.UserValue("userId").(int64)
		if !ok {
			log.Error(ErrUserIDConvertation)
			doJsonRes(ctx, fh.StatusInternalServerError, resp.New(false, nil, e.ErrInternalServer))
			return
		}

		// check bot exists
		existBot, err := db.CheckBotExist(userId, botId)
		if err != nil {
			log.Error(err)
			doJsonRes(ctx, fh.StatusInternalServerError, resp.New(false, nil, e.ErrInternalServer))
			return
		}
```

```
		if !existBot {
			doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil, e.ErrBotNotFound))
			return
		}

		// check bot component exists
		existComp, err := db.CheckComponentExist(botId, compId)
		if err != nil {
			log.Error(err)
			doJsonRes(ctx, fh.StatusInternalServerError, resp.New(false, nil, e.ErrInternalServer))
			return
		}

		if !existComp {
			doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil, e.ErrComponentNotFound))
			return
		}

		m := &model.Command{
			Type:       reqData.Type,
			Data:       reqData.Data,
			ComponentId: &compId,
			NextStepId:  nil,
			Status:     model.StatusCommandActive,
		}

		commandId, err := db.AddCommand(botId, m)
		if err != nil {
			log.Error(err)
			doJsonRes(ctx, fh.StatusInternalServerError, resp.New(false, nil, e.ErrInternalServer))
			return
		}

		// Invalidate component cache
		if err = r.DelComponent(botId, compId); err != nil {
			log.Error(err)
		}

		dataRes := &addCommandRes{
			Id: commandId,
		}

		doJsonRes(ctx, fh.StatusOK, resp.New(true, dataRes, nil))
	}
}


type setNextStepCommandReq struct {
	NextStepId *int64 `json:"nextStepId"`
}

func SetNextStepCommand(db *pgsql.Db, r *rdb.Rdb, log *zap.SugaredLogger) reqHandler {
	return func(ctx *fh.RequestCtx) {
		botId, err := strconv.ParseInt(ctx.UserValue("botId").(string), 10, 64)
		if err != nil {
			doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil, e.ErrInvalidRequest))
			return
		}
```

```
		compId, err := strconv.ParseInt(ctx.UserValue("compId").(string), 10, 64)
		if err != nil {
			doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil, e.ErrInvalidRequest))
			return
		}

		commandId, err := strconv.ParseInt(ctx.UserValue("commandId").(string), 10, 64)
		if err != nil {
			doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil, e.ErrInvalidRequest))
			return
		}

		var reqData setNextStepCommandReq
		if err = json.Unmarshal(ctx.PostBody(), &reqData); err != nil {
			doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil, e.ErrInvalidRequest))
			return
		}

		if reqData.NextStepId == nil {
			doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil,
e.MissingParam("nextStepId")))
			return
		}

		nextComponentId := reqData.NextStepId

		if *nextComponentId == compId {
			doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil, e.InvalidParam("nextStepId")))
			return
		}

		userId, ok := ctx.UserValue("userId").(int64)
		if !ok {
			log.Error(ErrUserIDConvertation)
			doJsonRes(ctx, fh.StatusInternalServerError, resp.New(false, nil, e.ErrInternalServer))
			return
		}

		// check bot exists
		existBot, err := db.CheckBotExist(userId, botId)
		if err != nil {
			log.Error(err)
			doJsonRes(ctx, fh.StatusInternalServerError, resp.New(false, nil, e.ErrInternalServer))
			return
		}

		if !existBot {
			doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil, e.ErrBotNotFound))
			return
		}

		// check bot component exists
		existInitialComp, err := db.CheckComponentExist(botId, compId)
		if err != nil {
			log.Error(err)
			doJsonRes(ctx, fh.StatusInternalServerError, resp.New(false, nil, e.ErrInternalServer))
			return
		}
```

```
		if !existInitialComp {
			doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil, e.ErrComponentNotFound))
			return
		}

		// Check next component exists
		existNextComp, err := db.CheckComponentExist(botId, *nextComponentId)
		if err != nil {
			log.Error(err)
			doJsonRes(ctx, fh.StatusInternalServerError, resp.New(false, nil, e.ErrInternalServer))
			return
		}

		if !existNextComp {
			doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil,
e.ErrNextComponentNotFound))
			return
		}

		// Check command exists
		existCommand, err := db.CheckCommandExist(botId, compId, commandId)
		if err != nil {
			log.Error(err)
			doJsonRes(ctx, fh.StatusInternalServerError, resp.New(false, nil, e.ErrInternalServer))
			return
		}

		if !existCommand {
			doJsonRes(ctx, fh.StatusBadRequest, resp.New(false, nil, e.ErrCommandNotFound))
			return
		}

		if err = db.SetNextStepCommand(botId, commandId, *nextComponentId); err != nil {
			log.Error(err)
			doJsonRes(ctx, fh.StatusInternalServerError, resp.New(false, nil, e.ErrInternalServer))
			return
		}

		// Invalidate component cache
		if err = r.DelComponent(botId, compId); err != nil {
			log.Error(err)
		}

		doJsonRes(ctx, fh.StatusOK, resp.New(true, nil, nil))
	}
}

package app

// WARN: bot not receive updates on app stop by panic

import (
	"os"
	"os/signal"
	"syscall"

	"github.com/redis/go-redis/v9"
	"github.com/valyala/fasthttp"
	"go.uber.org/zap"
```

```
	fastRouter "github.com/fasthttp/router"

	"github.com/botscubes/bot-service/internal/bot"
	"github.com/botscubes/bot-service/internal/config"
	"github.com/botscubes/bot-service/internal/database/pgsql"
	rdb "github.com/botscubes/bot-service/internal/database/redis"
	"github.com/botscubes/bot-service/internal/database/redisauth"
	"github.com/botscubes/user-service/pkg/token_storage"

	"github.com/mymmrac/telego"
)

type App struct {
	Router         *fastRouter.Router
	Server         *telego.MultiBotWebhookServer
	BotService     *bot.BotService
	Conf           *config.ServiceConfig
	Db             *pgsql.Db
	SessionStorage token_storage.TokenStorage
	RedisAuth      *redis.Client
	Redis          *rdb.Rdb
	Log            *zap.SugaredLogger
}

func (app *App) Run(logger *zap.SugaredLogger) error {
	var err error
	done := make(chan struct{}, 1)
	sigs := make(chan os.Signal, 1)
	signal.Notify(sigs, syscall.SIGINT, syscall.SIGTERM)

	app.Conf, err = config.GetConfig()
	if err != nil {
		return err
	}

	app.Log = logger

	app.Router = fastRouter.New()
	app.Server = &telego.MultiBotWebhookServer{
		Server: telego.FastHTTPWebhookServer{
			Server: &fasthttp.Server{
				Handler: app.Router.Handler,
			},
			Router: app.Router,
		},
	}

	app.BotService = bot.NewBotService(&app.Conf.Bot, app.Server)

	app.RedisAuth = redisauth.NewClient(&app.Conf.RedisAuth)
	app.SessionStorage = token_storage.NewRedisTokenStorage(app.RedisAuth)

	app.Redis = rdb.NewClient(&app.Conf.Redis)

	pgsqlUrl := "postgres://" + app.Conf.Pg.User + ":" + app.Conf.Pg.Pass + "@" + app.Conf.Pg.Host + ":" +
app.Conf.Pg.Port + "/" + app.Conf.Pg.Db
	if app.Db, err = pgsql.OpenConnection(pgsqlUrl); err != nil {
		return err
```

```
	}

	defer app.Db.CloseConnection()

	app.regiterHandlers()

	go func() {
		if err = app.Server.Start(app.Conf.Bot.ListenAddress); err != nil {
			app.Log.Error(err)
			sigs <- syscall.SIGTERM
		}
	}()

	// On close, error program
	go func() {
		<-sigs
		app.Log.Info("Stopping...")
		if err := app.BotService.StopBots(); err != nil {
			app.Log.Info("bots stop:\n", err)
		}
		done <- struct{}{}
	}()

	app.Log.Info("App Started")

	<-done

	return nil
}

package app

import (
	h "github.com/botscubes/bot-service/internal/api/handlers"
)

func (app *App) regiterHandlers() {
	app.Router.PanicHandler = h.PanicHandler(app.Log)

	app.Router.GET("/api/bots/health",
		h.Auth(h.Health, &app.SessionStorage, &app.Conf.JWTKey, app.Log))

	app.regBotsHandlers()
	app.regComponentsHandlers()
	app.regCommandsHandlers()
}

// Bot handlers
func (app *App) regBotsHandlers() {
	// Create new bot
	app.Router.POST("/api/bots",
		h.Auth(
			h.NewBot(app.Db, app.Log),
			&app.SessionStorage, &app.Conf.JWTKey, app.Log,
		))

	// Set bot token
	app.Router.POST("/api/bots/{botId}/token",
		h.Auth(
```

```
                        h.SetBotToken(app.Db, app.Log, app.BotService),
                        &app.SessionStorage, &app.Conf.JWTKey, app.Log,
                ))

        // Delete bot token
        app.Router.DELETE("/api/bots/{botId}/token",
                h.Auth(
                        h.DeleteBotToken(app.Db, app.Log, app.BotService),
                        &app.SessionStorage, &app.Conf.JWTKey, app.Log,
                ))

        // Start bot
        app.Router.PATCH("/api/bots/{botId}/start",
                h.Auth(
                        h.StartBot(app.Db, app.BotService, app.Redis, app.Log),
                        &app.SessionStorage, &app.Conf.JWTKey, app.Log,
                ))

        // Stop bot
        app.Router.PATCH("/api/bots/{botId}/stop",
                h.Auth(
                        h.StopBot(app.Db, app.BotService, app.Log),
                        &app.SessionStorage, &app.Conf.JWTKey, app.Log,
                ))

        // Get user bots
        app.Router.GET("/api/bots",
                h.Auth(
                        h.GetBots(app.Db, app.Log),
                        &app.SessionStorage, &app.Conf.JWTKey, app.Log,
                ))

        // Wipe bot data
        app.Router.PATCH("/api/bots/{botId}/wipe",
                h.Auth(
                        h.WipeBot(app.Db, app.Redis, app.BotService, app.Log),
                        &app.SessionStorage, &app.Conf.JWTKey, app.Log,
                ))
}

func (app *App) regComponentsHandlers() {
        // Adds a component to the bot structure
        app.Router.POST("/api/bots/{botId}/components",
                h.Auth(h.AddComponent(app.Db, app.Log), &app.SessionStorage, &app.Conf.JWTKey,
app.Log))

        // Delete bot component
        app.Router.DELETE("/api/bots/{botId}/components/{compId}",
                h.Auth(
                        h.DelComponent(app.Db, app.Redis, app.Log),
                        &app.SessionStorage, &app.Conf.JWTKey, app.Log,
                ))

        // Delete set of components
        app.Router.POST("/api/bots/{botId}/components/del",
                h.Auth(
                        h.DelSetOfComponents(app.Db, app.Redis, app.Log),
                        &app.SessionStorage, &app.Conf.JWTKey, app.Log,
                ))
```

```
	// update component
	app.Router.PATCH("/api/bots/{botId}/components/{compId}",
		h.Auth(
			h.UpdComponent(app.Db, app.Redis, app.Log),
			&app.SessionStorage, &app.Conf.JWTKey, app.Log,
		))

	// Set next step component
	app.Router.POST("/api/bots/{botId}/components/{compId}/next",
		h.Auth(
			h.SetNextStepComponent(app.Db, app.Redis, app.Log),
			&app.SessionStorage, &app.Conf.JWTKey, app.Log,
		))

	// Get bot components
	app.Router.GET("/api/bots/{botId}/components",
		h.Auth(
			h.GetBotComponents(app.Db, app.Log),
			&app.SessionStorage, &app.Conf.JWTKey, app.Log,
		))

	// Delete next step component
	app.Router.DELETE("/api/bots/{botId}/components/{compId}/next",
		h.Auth(
			h.DelNextStepComponent(app.Db, app.Redis, app.Log),
			&app.SessionStorage, &app.Conf.JWTKey, app.Log))
}

func (app *App) regCommandsHandlers() {
	// Add command
	app.Router.POST("/api/bots/{botId}/components/{compId}/commands",
		h.Auth(
			h.AddCommand(app.Db, app.Redis, app.Log),
			&app.SessionStorage, &app.Conf.JWTKey, app.Log,
		))

	// Delete component command
	app.Router.DELETE("/api/bots/{botId}/components/{compId}/commands/{commandId}",
		h.Auth(
			h.DelCommand(app.Db, app.Redis, app.Log),
			&app.SessionStorage, &app.Conf.JWTKey, app.Log,
		))

	// update command
	app.Router.PATCH("/api/bots/{botId}/components/{compId}/commands/{commandId}",
		h.Auth(
			h.UpdCommand(app.Db, app.Redis, app.Log),
			&app.SessionStorage, &app.Conf.JWTKey, app.Log,
		))

	// Set next step component command
	app.Router.POST("/api/bots/{botId}/components/{compId}/commands/{commandId}/next",
		h.Auth(
			h.SetNextStepCommand(app.Db, app.Redis, app.Log),
			&app.SessionStorage, &app.Conf.JWTKey, app.Log,
		))

	// Delete next step command
```

```
		app.Router.DELETE("/api/bots/{botId}/components/{compId}/commands/{commandId}/next",
			h.Auth(
				h.DelNextStepCommand(app.Db, app.Redis, app.Log),
				&app.SessionStorage, &app.Conf.JWTKey, app.Log,
			))
}

package bot

import (
	"errors"
	"strings"

	"github.com/botscubes/bot-service/internal/config"
	"github.com/botscubes/bot-service/internal/model"
	"github.com/mymmrac/telego"
	th "github.com/mymmrac/telego/telegohandler"
)

// Handles incoming Message & EditedMessage from Telegram.
func (btx *TBot) messageHandler() th.Handler {
	return func(bot *telego.Bot, update telego.Update) {
		var message *telego.Message
		if update.Message != nil {
			message = update.Message
		} else {
			message = update.EditedMessage
		}

		// Get user stepID
		stepID, err := btx.getUserStep(message.From)
		if err != nil {
			return
		}

		// find next component for execute
		ok, component, nextStepId := btx.findComponent(stepID, message)
		if !ok {
			return
		}

		if nextStepId != stepID {
			if err := btx.Rdb.SetUserStep(btx.Id, message.From.ID, nextStepId); err != nil {
				btx.log.Error(err)
			}
			// Async upd stepID in db
			go btx.setUserStep(message.From.ID, nextStepId)
		}

		if err := btx.execMethod(bot, message, component); err != nil {
			btx.log.Error(err)
		}
	}
}

// Handles incoming Message with command (eg. /start) from Telegram.
// So far only /start command !!!
func (btx *TBot) commandHandler() th.Handler {
	return func(bot *telego.Bot, update telego.Update) {
```

```
                message := update.Message

                if !commandEqual(message.Text, "start") {
                        return
                }

                stepID := int64(config.MainComponentId)

                // find next component for execute
                ok, component, nextStepId := btx.findComponent(stepID, message)
                if !ok {
                        return
                }

                if nextStepId != stepID {
                        if err := btx.Rdb.SetUserStep(btx.Id, message.From.ID, nextStepId); err != nil {
                                btx.log.Error(err)
                        }
                        // Async upd stepID in db
                        go btx.setUserStep(message.From.ID, nextStepId)
                }

                if err := btx.execMethod(bot, message, component); err != nil {
                        btx.log.Error(err)
                }
        }
}

// Determining the next step in the bot structure
func (btx *TBot) findComponent(stepID int64, message *telego.Message) (bool, *model.Component, int64) {
        var origComponent *model.Component
        var component *model.Component
        origStepID := stepID

        // for cycle detect
        stepsPassed := make(map[int64]struct{})
        isFound := false

        for {
                // This part of the loop (before the "isFound" condition) is used to automatically
                // skip the starting component and undefined components.
                // Also, the next component is selected here by the id found in the second part of the cycle.

                if _, ok := stepsPassed[stepID]; ok {
                        if origStepID == stepID {
                                break
                        }

                        btx.log.Warnf("cycle detected: bot #%d", btx.Id)
                        return false, nil, 0
                }

                stepsPassed[stepID] = struct{}{}

                var err error
                component, err = btx.getComponent(stepID)
                if err != nil {
                        if errors.Is(err, ErrNotFound) {
                                stepID = config.MainComponentId
```

```
                        continue
                }

                return false, nil, 0
        }

        if origComponent == nil {
                origComponent = component
        }

        // check main component
        if component.IsMain {
                if component.NextStepId == nil || *component.NextStepId == stepID {
                        btx.log.Warnf("error referring to the next component in the main component: bot
#%d", btx.Id)
                        return false, nil, 0
                }

                stepID = *component.NextStepId
                isFound = true
                continue
        }

        if isFound {
                // next component was found successfully
                break
        }

        // In this part, the id of the next component is determined.
        // In case of successful identification of the ID, an additional check occurs in the first part of the
cycle.

        isFound = true

        if component.NextStepId != nil {
                stepID = *component.NextStepId
                continue
        }

        command := findComponentCommand(&message.Text, component.Commands)
        if command != nil && command.NextStepId != nil {
                stepID = *command.NextStepId
                continue
        }

        // next component not found, will be executed initial (current) component
        component = origComponent
        stepID = origStepID
        break
    }

    return true, component, stepID
}

func (btx *TBot) execMethod(bot *telego.Bot, message *telego.Message, component *model.Component) error {
    switch *component.Data.Type {
    case "text":
        if err := sendMessage(bot, message, component); err != nil {
            return err
```

```
                }
        default:
                btx.log.Warn("Unknown type method: ", *component.Data.Type)
        }

        return nil
}

// Determine commnad by !message text!
func findComponentCommand(mes *string, commands *model.Commands) *model.Command {
        // work with command type - text
        for _, command := range *commands {
                // The comparison is not case sensitive
                if strings.EqualFold(*command.Data, *mes) {
                        return command
                }
        }

        return nil
}

func commandEqual(messageText string, command string) bool {
        matches := th.CommandRegexp.FindStringSubmatch(messageText)
        if len(matches) != th.CommandMatchGroupsLen {
                return false
        }

        return strings.EqualFold(matches[1], command)
}

package bot

import (
        "errors"

        "github.com/botscubes/bot-service/internal/config"
        rdb "github.com/botscubes/bot-service/internal/database/redis"
        "github.com/botscubes/bot-service/internal/model"
        "github.com/mymmrac/telego"
)

func (btx *TBot) getUserStep(from *telego.User) (int64, error) {
        // try get userStep from cache
        stepID, err := btx.Rdb.GetUserStep(btx.Id, from.ID)
        if err == nil {
                return stepID, nil
        }

        if !errors.Is(err, rdb.ErrNotFound) {
                btx.log.Error(err)
        }

        // userStep not found in cache, try get from db
        stepID, err = btx.Db.UserStepByTgId(btx.Id, from.ID)
        if err != nil {
                btx.log.Error(err)
                return 0, err
        }
```

```
		if err = btx.Rdb.SetUserStep(btx.Id, from.ID, stepID); err != nil {
			btx.log.Error(err)
		}

		return stepID, nil
	}

	func (btx *TBot) getComponent(stepID int64) (*model.Component, error) {
		// try get component from cache
		component, err := btx.Rdb.GetComponent(btx.Id, stepID)
		if err == nil {
			return component, nil
		}

		if !errors.Is(err, rdb.ErrNotFound) {
			btx.log.Error(err)
		}

		// check bot components exists in cache
		ex, err := btx.Rdb.CheckComponentsExist(btx.Id)
		if err != nil {
			btx.log.Error(err)
		}

		// components not found in cache
		if err == nil && ex == 0 {
			// get all components from db
			components, err := btx.Db.ComponentsForBot(btx.Id)
			if err != nil {
				btx.log.Error(err)
			}

			if err := btx.Rdb.SetComponents(btx.Id, components); err != nil {
				btx.log.Error(err)
			}

			component, err := btx.Rdb.GetComponent(btx.Id, stepID)
			if err == nil {
				return component, nil
			}

			if !errors.Is(err, rdb.ErrNotFound) {
				btx.log.Error(err)
			}
		}

		// component not found in cache, try get from db
		exist, err := btx.Db.CheckComponentExist(btx.Id, stepID)
		if err != nil {
			btx.log.Error(err)
			return nil, err
		}

		if exist {
			component, err = btx.Db.ComponentForBot(btx.Id, stepID)
			if err != nil {
				btx.log.Error(err)
				return nil, err
			}
```

```
                if err = btx.Rdb.SetComponent(btx.Id, component); err != nil {
                        btx.log.Error(err)
                }

                return component, nil
        }

        return nil, ErrNotFound
}

func (btx *TBot) setUserStep(userId int64, stepID int64) {
        if err := btx.Db.SetUserStepByTgId(btx.Id, userId, stepID); err != nil {
                btx.log.Error(err)
        }
}
package redis

import (
        "context"
        "errors"
        "strconv"

        "github.com/botscubes/bot-service/internal/config"
        "github.com/redis/go-redis/v9"
)

func (rdb *Rdb) SetUserStep(botId int64, userID int64, stepID int64) error {
        ctx := context.Background()

        key := "bot" + strconv.FormatInt(botId, 10) + ":user:" + strconv.FormatInt(userID, 10)

        if err := rdb.HSet(ctx, key, "step", stepID).Err(); err != nil {
                return err
        }

        return rdb.Expire(ctx, key, config.RedisExpire).Err()
}

func (rdb *Rdb) GetUserStep(botId int64, userID int64) (int64, error) {
        ctx := context.Background()

        key := "bot" + strconv.FormatInt(botId, 10) + ":user:" + strconv.FormatInt(userID, 10)

        var stepID int64
        err := rdb.HGet(ctx, key, "step").Scan(&stepID)
        if err != nil && !errors.Is(err, redis.Nil) {
                return 0, err
        }

        if errors.Is(err, redis.Nil) {
                return 0, ErrNotFound
        }

        return stepID, nil
}

func (rdb *Rdb) CheckUserExist(botId int64, userID int64) (int64, error) {
        ctx := context.Background()
```

```
	key := "bot" + strconv.FormatInt(botId, 10) + ":user:" + strconv.FormatInt(userID, 10)
	return rdb.Exists(ctx, key).Result()
}
package redis

import (
	"context"
	"errors"
	"strconv"

	"github.com/botscubes/bot-service/internal/config"
	"github.com/botscubes/bot-service/internal/model"
	"github.com/redis/go-redis/v9"
)

func (rdb *Rdb) SetComponent(botId int64, comp *model.Component) error {
	ctx := context.Background()

	key := "bot" + strconv.FormatInt(botId, 10) + ":component"

	if err := rdb.HSet(ctx, key, strconv.FormatInt(comp.Id, 10), comp).Err(); err != nil {
		return err
	}

	return rdb.Expire(ctx, key, config.RedisExpire).Err()
}

func (rdb *Rdb) SetComponents(botId int64, comps *[]*model.Component) error {
	for _, v := range *comps {
		if err := rdb.SetComponent(botId, v); err != nil {
			return err
		}
	}

	return nil
}

func (rdb *Rdb) GetComponent(botId int64, compId int64) (*model.Component, error) {
	ctx := context.Background()

	key := "bot" + strconv.FormatInt(botId, 10) + ":component"

	component := &model.Component{}

	result, err := rdb.HGet(ctx, key, strconv.FormatInt(compId, 10)).Result()
	if err != nil && !errors.Is(err, redis.Nil) {
		return nil, err
	}

	if result == "" {
		return nil, ErrNotFound
	}

	if err := component.UnmarshalBinary([]byte(result)); err != nil {
		return nil, err
	}

	return component, nil
}
```

```
func (rdb *Rdb) DelComponent(botId int64, compId int64) error {
	ctx := context.Background()
	key := "bot" + strconv.FormatInt(botId, 10) + ":component"
	return rdb.HDel(ctx, key, strconv.FormatInt(compId, 10)).Err()
}

func (rdb *Rdb) CheckComponentsExist(botId int64) (int64, error) {
	ctx := context.Background()
	key := "bot" + strconv.FormatInt(botId, 10) + ":component"
	return rdb.Exists(ctx, key).Result()
}
```

```
package pgsql

import (
	"context"
	"strconv"

	"github.com/botscubes/bot-service/internal/config"
	"github.com/botscubes/bot-service/internal/model"
)

func (db *Db) AddComponent(botId int64, m *model.Component) (int64, error) {
	var id int64
	query := `INSERT INTO ` + prefixSchema + strconv.FormatInt(botId, 10) + `.component
				("data", keyboard, next_step_id, is_main,"position", status) VALUES ($1, $2, $3, $4, $5,
$6) RETURNING id;`

	if err := db.Pool.QueryRow(
		context.Background(), query, m.Data, m.Keyboard, m.NextStepId, m.IsMain, m.Position,
m.Status,
	).Scan(&id); err != nil {
		return 0, err
	}

	return id, nil
}

func (db *Db) CheckComponentExist(botId int64, compId int64) (bool, error) {
	var c bool
	query := `SELECT EXISTS(SELECT 1 FROM ` + prefixSchema + strconv.FormatInt(botId, 10) +
`.component
				WHERE id = $1 AND status = $2) AS "exists";`

	if err := db.Pool.QueryRow(
		context.Background(), query, compId, model.StatusComponentActive,
	).Scan(&c); err != nil {
		return false, err
	}

	return c, nil
}

func (db *Db) SetNextStepComponent(botId int64, compId int64, nextStepId int64) error {
	query := `UPDATE ` + prefixSchema + strconv.FormatInt(botId, 10) + `.component
				SET next_step_id = $1 WHERE id = $2;`

	_, err := db.Pool.Exec(context.Background(), query, nextStepId, compId)
```

```
	return err
}

func (db *Db) ComponentsForEd(botId int64) (*[]*model.Component, error) {
	var data []*model.Component

	query := `SELECT id, data, keyboard, ARRAY(
				SELECT jsonb_build_object('id', id, 'data', data, 'type', type, 'componentId',
component_id, 'nextStepId', next_step_id)
				FROM ` + prefixSchema + strconv.FormatInt(botId, 10) + `.command
				WHERE component_id = t.id AND status = $1 ORDER BY id
			), next_step_id, is_main, position, status
			FROM ` + prefixSchema + strconv.FormatInt(botId, 10) + `.component t
			WHERE status = $2 ORDER BY id;`

	rows, err := db.Pool.Query(context.Background(), query, model.StatusCommandActive,
model.StatusComponentActive)
	if err != nil {
		return nil, err
	}

	// WARN: status not used
	for rows.Next() {
		var r model.Component
		r.Commands = &model.Commands{}

		if err = rows.Scan(&r.Id, &r.Data, &r.Keyboard, r.Commands, &r.NextStepId, &r.IsMain,
&r.Position, &r.Status); err != nil {
			return nil, err
		}

		data = append(data, &r)
	}

	if rows.Err() != nil {
		return nil, rows.Err()
	}

	return &data, nil
}

func (db *Db) DelNextStepComponent(botId int64, compId int64) error {
	query := `UPDATE ` + prefixSchema + strconv.FormatInt(botId, 10) + `.component
			SET next_step_id = null WHERE id = $1;`

	_, err := db.Pool.Exec(context.Background(), query, compId)
	return err
}

func (db *Db) DelNextStepComponentByNS(botId int64, nextStep int64) error {
	query := `UPDATE ` + prefixSchema + strconv.FormatInt(botId, 10) + `.component
			SET next_step_id = null WHERE next_step_id = $1;`

	_, err := db.Pool.Exec(context.Background(), query, nextStep)
	return err
}

func (db *Db) DelNextStepsComponentByNS(botId int64, nextSteps *[]int64) error {
	query := `UPDATE ` + prefixSchema + strconv.FormatInt(botId, 10) + `.component
```

```
                        SET next_step_id = null WHERE next_step_id = ANY($1::bigint[]);`

        _, err := db.Pool.Exec(context.Background(), query, nextSteps)
        return err
}

func (db *Db) DelComponent(botId int64, compId int64) error {
        query := `UPDATE ` + prefixSchema + strconv.FormatInt(botId, 10) + `.component
                        SET status = $1 WHERE id = $2;`

        _, err := db.Pool.Exec(context.Background(), query, model.StatusComponentDel, compId)
        return err
}

func (db *Db) DelSetOfComponents(botId int64, data *[]int64) error {
        query := `UPDATE ` + prefixSchema + strconv.FormatInt(botId, 10) + `.component
                        SET status = $1 WHERE id = ANY($2::bigint[]);`

        _, err := db.Pool.Exec(context.Background(), query, model.StatusComponentDel, data)
        return err
}

func (db *Db) DelAllComponents(botId int64) error {
        query := `UPDATE ` + prefixSchema + strconv.FormatInt(botId, 10) + `.component
                        SET status = $1 WHERE id != $2;`

        _, err := db.Pool.Exec(context.Background(), query, model.StatusComponentDel,
config.MainComponentId)
        return err
}

func (db *Db) ComponentsForBot(botId int64) (*[]*model.Component, error) {
        // TODO: REMOVE POSITION !
        // Change return value to *map[int64]*model.Component???

        var data []*model.Component

        query := `SELECT id, data, keyboard, ARRAY(
                        SELECT jsonb_build_object('id', id, 'data', data, 'type', type, 'componentId',
component_id, 'nextStepId', next_step_id)
                                FROM ` + prefixSchema + strconv.FormatInt(botId, 10) + `.command
                                WHERE component_id = t.id AND status = $1 ORDER BY id
                        ), next_step_id, is_main, position, status
                        FROM ` + prefixSchema + strconv.FormatInt(botId, 10) + `.component t
                        WHERE status = $2 ORDER BY id;`

        rows, err := db.Pool.Query(context.Background(), query, model.StatusCommandActive,
model.StatusComponentActive)
        if err != nil {
                return nil, err
        }

        // WARN: status not used
        for rows.Next() {
                var r model.Component
                r.Commands = &model.Commands{}
                if err = rows.Scan(&r.Id, &r.Data, &r.Keyboard, r.Commands, &r.NextStepId, &r.IsMain,
&r.Position, &r.Status); err != nil {
                        return nil, err
```

```
			}

			data = append(data, &r)
		}

		if rows.Err() != nil {
			return nil, rows.Err()
		}

		return &data, nil
}

func (db *Db) ComponentForBot(botId int64, compID int64) (*model.Component, error) {
		// TODO: REMOVE POSITION !

		prefix := prefixSchema + strconv.FormatInt(botId, 10)

		query := `SELECT id, data, keyboard, ARRAY(
			SELECT jsonb_build_object('id', id, 'data', data, 'type', type, 'componentId', component_id,
'nextStepId', next_step_id)
			FROM ` + prefix + `.command
			WHERE component_id = t.id AND status = $1 ORDER BY id
		), next_step_id, is_main, position, status
		FROM ` + prefix + `.component t
		WHERE status = $2 AND id = $3 ORDER BY id;`

		// WARN: status not used
		var r model.Component
		r.Commands = &model.Commands{}
		if err := db.Pool.QueryRow(
			context.Background(), query, model.StatusCommandActive, model.StatusComponentActive,
compID,
		).Scan(&r.Id, &r.Data, &r.Keyboard, r.Commands, &r.NextStepId, &r.IsMain, &r.Position, &r.Status); err
!= nil {
			return nil, err
		}

		return &r, nil
}

func (db *Db) UpdComponentPosition(botId int64, compId int64, pos *model.Point) error {
		prefix := prefixSchema + strconv.FormatInt(botId, 10)
		query := `UPDATE ` + prefix + `.component SET "position" = $1 WHERE id = $2;`

		_, err := db.Pool.Exec(context.Background(), query, pos, compId)
		return err
}

func (db *Db) UpdComponentData(botId int64, compId int64, data *model.Data) error {
		prefix := prefixSchema + strconv.FormatInt(botId, 10)
		query := `UPDATE ` + prefix + `.component SET "data" = $1 WHERE id = $2;`

		_, err := db.Pool.Exec(context.Background(), query, data, compId)
		return err
}
```

Приложение Б

(обязательное)

Текст конфигурационного файла Docker Compose

```
version: '3'

services:
  nginx:
    image: nginx:alpine
    restart: unless-stopped
    env_file:
      - ./config/env/nginx.env
    volumes:
      - ./data/nginx/templates:/etc/nginx/templates:ro
      - ./log/nginx:/var/log/nginx:rw
      - ./data/certbot/conf:/etc/letsencrypt:rw
      - ./data/certbot/www:/var/www/certbot:rw
    ports:
      - "80:80"
      - "443:443"
    depends_on:
      - bot
    command: "/bin/sh -c '/docker-entrypoint.sh nginx -g \"daemon off;\"; while :; do sleep 6h & wait $${!}; nginx -s
reload; done & nginx -g \"daemon off;\";'"
    networks:
      - bot_nw
      - user_nw
      - frontend_nw

  certbot:
    image: certbot/certbot
    restart: unless-stopped
    volumes:
      - ./data/certbot/conf:/etc/letsencrypt:rw
      - ./data/certbot/www:/var/www/certbot:rw
    entrypoint: "/bin/sh -c 'trap exit TERM; while :; do certbot renew; sleep 12h & wait $${!}; done;'"

  bot:
    build: ../bot-service
    restart: unless-stopped
    logging:
      driver: "json-file"
      options:
        max-size: "200k"
        max-file: "5"
    env_file:
      - ./config/env/bot.env
      - ./config/env/pgsql_bot.env
      - ./config/env/redis_auth.env
      - ./config/env/redis_bot.env
    volumes:
      - ../bot-service:/app:rw
      - /root/go/pkg/mod/cache:/go/pkg/mod/cache:rw
    tty: false
    command: go run ./cmd/bot/main.go
    networks:
      - bot_nw
```

```
    - redis_auth_nw

  pgsql_bot:
    image: postgres:15.1-alpine
    restart: unless-stopped
    env_file:
      - ./config/env/pgsql_bot.env
    volumes:
      - ./data/pgsql_bot/pginit:/docker-entrypoint-initdb.d:rw
      - ./data/pgsql_bot/pgdata:/var/lib/postgresql/data:rw
    ports:
      - "5431:5432"
    networks:
      - bot_nw

  user:
    build: ../user-service
    restart: unless-stopped
    volumes:
      - ../user-service:/app:rw
      - /root/go/pkg/mod/cache:/go/pkg/mod/cache:rw
    tty: false
    command: go run ./cmd/server.go
    networks:
      - user_nw
      - redis_auth_nw

  pgsql_user:
    image: postgres:15.1-alpine
    restart: unless-stopped
    env_file:
      - ./config/env/pgsql_user.env
    volumes:
      - ./data/pgsql_user/pginit:/docker-entrypoint-initdb.d:rw
      - ./data/pgsql_user/pgdata:/var/lib/postgresql/data:rw
    ports:
      - "5430:5432"
    networks:
      - user_nw

  redis_auth:
    image: redis:7.0.5-alpine
    command: redis-server
    volumes:
      - ./data/redis_auth/redis_data:/var/lib/redis:rw
      - ./config/redis_auth/redis.conf:/usr/local/etc/redis/redis.conf:ro
    networks:
      - redis_auth_nw

  redis_bot:
    image: redis:7.0.5-alpine
    command: redis-server
    volumes:
      - ./data/redis_bot/redis_data:/var/lib/redis:rw
      - ./config/redis_bot/redis.conf:/usr/local/etc/redis/redis.conf:ro
    networks:
      - bot_nw

  web:
```

```
    container_name: web-client
    build: ../web-client/editor
    networks:
      - frontend_nw

networks:
  bot_nw:
  user_nw:
  redis_auth_nw:
  frontend_nw:
```

Приложение В

(Обязательное)

Перечень сокращений

БД – база данных

СУБД – система управления базами данных

JSON – JavaScript Object Notation

ER- entity relationship

SQL - Structured Query Language

API - Application Programming Interface

REST - Representational State Transfer

Приложение Г

(справочное)

Библиографический список

1. HTTP – Википедия [Электронный ресурс]. – Режим доступа: https://ru.wikipedia.org/wiki/HTTP.
2. JSON – Википедия [Электронный ресурс]. – Режим доступа: https://ru.wikipedia.org/wiki/JSON.
3. Telegram Bot API [Электронный ресурс]. – Режим доступа: https://core.telegram.org/bots/api.
4. Documentation – The Go Programming Language [Электронный ресурс]. – Режим доступа: https://go.dev/doc/.
5. PostgreSQL – Documentation [Электронный ресурс]. – Режим доступа: https://www.postgresql.org/docs/.
6. Documentation | Redis [Электронный ресурс]. – Режим доступа: https://redis.io/docs/.
7. Overview | Docker Documentation [Электронный ресурс]. – Режим доступа: https://docs.docker.com/get-started/