«Разработка клиент серверного приложения для хранения данных в облаке»

**Оглавление**

[**Введение** 3](#_Toc188553593)

[**1. ОПИСАНИЕ ТЕХНОЛОГИЙ** 5](#_Toc188553594)

[**1.1) Описание технологии REST API** 5](#_Toc188553595)

[**1.2) Описание технологии Телеграм-бот** 6](#_Toc188553596)

[**1.3) Описание технологии Docker** 7](#_Toc188553597)

[**1.4) Описание технологии PostgreSQL** 9](#_Toc188553598)

[**1.5) Сравнительная характеристика Java** 11](#_Toc188553599)

[**1.6) Характеристики Java** 1](#_Toc188553600)2

[**1.5) Сравнительная характеристика Go**](#_Toc188553599) 13

[**1.6) Характеристики Go** 1](#_Toc188553600)4

[**1.7) Клиент-серверная архитектура**](#_Toc188553601) 15

[**2. РАЗРАБОТКА ПРИЛОЖЕНИЯ**](#_Toc188553602) 16

[**2.1) Структура проекта**](#_Toc188553603) 16

[**2.2) Разработка серверной части**](#_Toc188553604) 18

[**2.3) Разаработка клиентской части**](#_Toc188553605) 33

**2.4)** [**Инструкция для запуска**](#_Toc188553606) 42

**2.5)** [**Инструкция пользователя**](#_Toc188553607) 44

[**Выводы**](#_Toc188553608) 47

[**Список используемых источников**](#_Toc188553609) 48

[**Приложение**](#_Toc188553610) 49

# **Введение**

В современном цифровом мире, где доступ к данным и их хранение играют ключевую роль, облачные хранилища становятся незаменимым инструментом для пользователей и организаций. Они предоставляют возможность безопасного хранения, совместного использования и удаленного доступа к данным из любой точки мира, обеспечивая удобство и надежность.

Главная цель данного проекта заключается в разработке удобного приложения для работы с облачным хранилищем, поддерживающего как интерфейс командной строки (CLI), так и графический интерфейс (GUI). Это приложение позволит пользователям эффективно управлять своими данными, синхронизировать файлы между устройствами и безопасно делиться ими с другими.

Актуальность использования клиент-серверной архитектуры для хранения данных в облаке с использованием REST API и клиента на Telegram обусловлена несколькими ключевыми факторами, отражающими текущие тенденции в области технологий и потребности пользователей.

Гибкость и масштабируемость: Использование REST API для обмена данными между клиентом и сервером обеспечивает гибкость и масштабируемость решения. Архитектура REST является стандартом для большинства веб-сервисов, что позволяет интегрировать облачное хранилище с различными платформами и сервисами. Архитектура без состояния (stateless) REST API позволяет легко масштабировать систему, добавлять новые серверы или изменять логику без значительных изменений в клиентской части.

Доступность и мобильность: Использование Telegram как клиента дает пользователям доступ к сервисам и данным в любое время и из любого места. Telegram является одним из самых популярных мессенджеров, доступным на разных устройствах (мобильные телефоны, ПК, планшеты), что делает взаимодействие с сервером удобным и универсальным. Телеграм-боты позволяют пользователям взаимодействовать с облачным хранилищем через привычный интерфейс мессенджера, что упрощает использование системы и не требует установки дополнительного программного обеспечения.

Безопасность и простота авторизации: Telegram обеспечивает встроенные механизмы безопасности, такие как двухфакторная аутентификация и безопасное подключение через TLS. Взаимодействие с облачным хранилищем через Telegram-бота позволяет использовать эти механизмы для безопасного доступа к данным. Также использование Telegram-бота упрощает процесс авторизации и регистрации пользователей, избавляя от необходимости разрабатывать сложные системы логинов и паролей.

Интерактивность и уведомления: Клиент на Telegram позволяет создавать интерактивные интерфейсы с кнопками, меню и формами, что значительно улучшает пользовательский опыт при взаимодействии с облачным хранилищем. Бот может отправлять уведомления о событиях, таких как загрузка или удаление файлов, завершение операций, ошибки и другие важные события, что повышает удобство использования.

Экономия ресурсов и высокая производительность: Использование REST API для связи с сервером позволяет эффективно управлять запросами и оптимизировать использование серверных ресурсов. Легковесные запросы REST API, направленные на получение, изменение или удаление данных, обеспечивают низкую задержку и высокую производительность, что особенно важно при работе с большими объемами данных в облаке.

Автоматизация и интеграция с другими сервисами: Телеграм-боты, интегрированные с REST API, могут автоматизировать множество процессов, таких как синхронизация данных, создание отчетов, управление файлами и т.д. Они также могут быть связаны с другими внешними сервисами, расширяя функциональность облачного хранилища и предоставляя пользователю более широкие возможности для работы с данными.

Таким образом, использование клиент-серверной архитектуры для хранения данных в облаке с REST API и клиентом на Telegram представляет собой современное и эффективное решение, которое сочетает гибкость, удобство, безопасность и масштабируемость. Это подход, который отвечает требованиям пользователей и бизнеса в условиях растущих объемов данных и необходимости быстрого и безопасного доступа к ним.

**ГЛАВА 1. ОПИСАНИЕ ТЕХНОЛОГИЙ**

## **1.1) Описание технологии REST API**

REST (Representational State Transfer) — это архитектурный стиль, широко используемый для создания распределенных веб-приложений и взаимодействия между клиентами и серверами через Интернет. Он был предложен Роем Филдингом в 2000 году как часть его докторской диссертации и с тех пор стал основой большинства современных веб-сервисов. REST основывается на простоте, масштабируемости и использовании стандартных протоколов, таких как HTTP, что делает его удобным и эффективным для интеграции различных систем.

Ключевыми принципами REST являются использование ресурсов, доступ к которым осуществляется через URI, и стандартизированные методы HTTP (GET, POST, PUT, DELETE). Эти принципы делают REST понятным и универсальным для создания веб-API, обеспечивающих гибкость и простоту взаимодействия между различными приложениями и платформами.

**Основные Концепции REST API**

Основные концепции REST включают несколько ключевых принципов. Все данные представляются как ресурсы, доступные через уникальные URI. Для работы с ресурсами используются стандартные методы HTTP, такие как GET для получения данных, POST для создания, PUT для обновления и DELETE для удаления. Ответы сервера сопровождаются статус-кодами HTTP, которые отражают результат операции, например, 200 OK или 404 Not Found. REST следует принципу без состояния (stateless), где каждое обращение клиента к серверу полностью автономно и не требует хранения состояния на сервере. Также REST поддерживает кэширование, что позволяет ускорять обработку запросов и снижать нагрузку на сервер.

**Компоненты REST-сервера**

REST включает в себя базу данных, которая хранит данные, которые нужно хранить и которые используются для корректной работы сервиса и обработчик HTTP запросов, который обеспечивает обработку приходящих запросов и данных из этих запросов для последующего взаимодействия с базой данных.

**Недостатки REST API**

Несмотря на популярность, REST API имеет свои недостатки. Одним из главных является избыточность данных, поскольку каждый запрос часто включает все данные ресурса, даже если нужна только их часть. Это может привести к увеличению объема передаваемых данных и замедлению работы на медленных соединениях. REST не всегда подходит для сложных взаимосвязей между ресурсами, так как для получения связанных данных требуется несколько запросов. Еще одним недостатком является отсутствие строгого стандарта, что может привести к непоследовательности реализации API у разных разработчиков. Также REST неэффективен для приложений, требующих постоянного взаимодействия в реальном времени, так как он основывается на запрос-ответной модели, а не на постоянном соединении.

**Преимущества REST API**

REST API обладает несколькими ключевыми преимуществами. Во-первых, это простота и удобство использования. REST основывается на стандартных HTTP методах (GET, POST, PUT, DELETE), которые знакомы большинству разработчиков, что упрощает разработку и поддержку. Во-вторых, REST имеет масштабируемость: архитектура без состояния (stateless) позволяет легко масштабировать серверы, так как каждый запрос полностью независим и не требует сохранения информации о предыдущих взаимодействиях. Кроме того, REST поддерживает кэширование, что улучшает производительность и снижает нагрузку на серверы, ускоряя обработку запросов. Еще одним преимуществом является гибкость: REST API можно использовать для взаимодействия между различными платформами и системами, что делает его универсальным и подходящим для различных типов приложений. Кроме того, REST прост в интеграции с другими веб-технологиями и легко поддерживает работы с форматами данных, такими как JSON и XML, что делает его идеальным для современных веб-приложений.

## **1.2) Описание технологии Телеграм-бот**

Телеграм-бот — это специальная программа, которая работает внутри мессенджера Telegram и автоматизирует выполнение различных задач. Боты могут взаимодействовать с пользователями, отвечать на команды, отправлять сообщения, обрабатывать мультимедийные файлы, а также интегрироваться с внешними сервисами и API. Для работы с ботами используется Telegram Bot API, который предоставляет разработчикам набор инструментов для создания и управления ботами.

Телеграм-боты могут выполнять разнообразные функции, например, напоминания, чат-боты для поддержки клиентов, боты для автоматизации рабочих процессов, получение новостей, выполнение команд и взаимодействие с другими платформами. Они могут работать в фоновом режиме, предоставляя пользователю автоматизированный сервис без необходимости постоянного вмешательства. Боты могут отвечать на текстовые сообщения, команды, а также интегрировать кнопки и формы для более сложного взаимодействия.

Для того чтобы использовать телеграм-бота, пользователю не нужно устанавливать дополнительное программное обеспечение — достаточно обычного приложения Telegram. Боты могут быть доступны через поиск в Telegram, а взаимодействие с ними обычно происходит через простые команды или меню.

**Преимущества Телеграм бота**

Схема работы TelegramAPI имеед такой вид

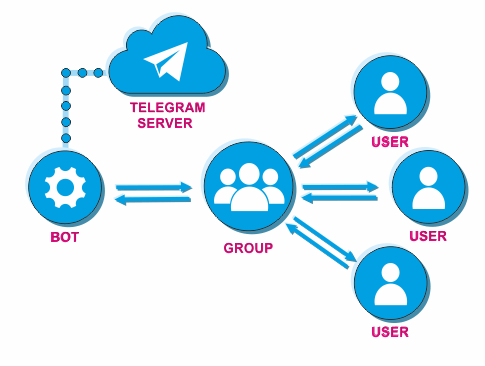


Рисунок 1. Принцип работы Telegram бота

Во-первых, Telegram — это одна из самых популярных платформ для обмена сообщениями с миллионами активных пользователей по всему миру, что дает боту доступ к широкой аудитории без необходимости разрабатывать отдельное приложение. Во-вторых, Telegram предоставляет удобный и гибкий API, который позволяет легко создавать ботов с разными функциями, включая отправку сообщений, обработку команд и работу с мультимедиа.

Кроме того, Telegram-боты могут работать в фоновом режиме, не требуя от пользователя постоянного внимания, что делает их идеальными для автоматизированных сервисов. Также они поддерживают интеграцию с другими сервисами и позволяют быстро настроить систему уведомлений и оповещений. Еще одним преимуществом является безопасность и простота авторизации через Telegram, что избавляет от необходимости разрабатывать сложные системы регистрации и входа в систему. Наконец, Telegram-боты обеспечивают доступность через различные устройства (мобильные телефоны, ПК, веб-версии), что дает пользователям гибкость и удобство в использовании.

## **1.3) Описание технологии Docker**

Docker — это платформа для автоматизации развертывания, масштабирования и управления приложениями в контейнерах. Контейнеры представляют собой легковесные, изолированные окружения, которые позволяют запускать приложения с их зависимостями на любом сервере или в облаке, независимо от операционной системы и аппаратного обеспечения. Основной идеей Docker является упаковка приложений и их зависимостей в контейнеры, которые можно легко перемещать, запускать и масштабировать.

Docker работает ближе к операционной системе, что позволяет экономить ресурсы компьютера

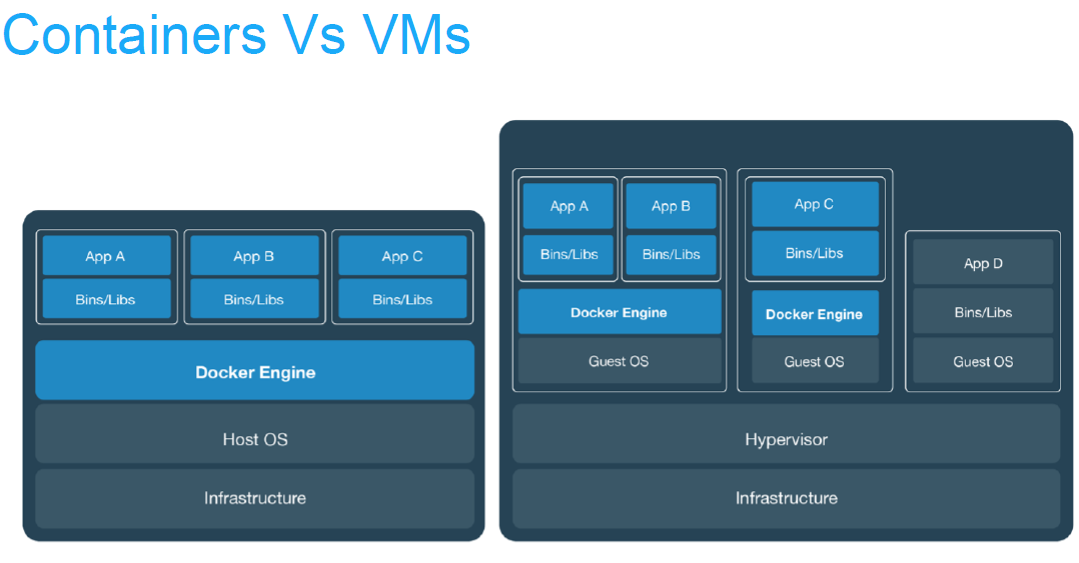


Рисунок 2. Сравнение принципа работы докер контейнеров и виртуальных машин

Docker предоставляет несколько ключевых компонентов. Docker Engine — это основа, которая управляет контейнерами, создавая, запускавая и останавливая их. Docker Hub и другие реестры — это репозитории, в которых хранятся образы контейнеров, которые можно загрузить и использовать для создания контейнеров. Docker Compose позволяет легко управлять многоконтейнерными приложениями, описывая их конфигурацию в виде YAML-файла.

С Docker разработчики могут создавать приложения в виде контейнеров, которые включают все необходимые библиотеки и инструменты, что решает проблему несовместимости версий между средами разработки, тестирования и продакшн. Это упрощает процесс развертывания и уменьшает вероятность возникновения ошибок, связанных с различиями в конфигурациях. Docker также способствует более быстрому и эффективному использованию ресурсов, поскольку контейнеры используют те же ядра операционных систем, в отличие от виртуальных машин, которые требуют собственных операционных систем и имеют более высокие накладные расходы.

**Преимущества Docker**

Docker предлагает несколько значительных преимуществ. Он обеспечивает портируемость, так как контейнеры могут быть запущены на любой системе, поддерживающей Docker, независимо от ОС и оборудования. Благодаря изоляции контейнеров, приложения не конфликтуют друг с другом, упрощая управление зависимостями. Docker также легковесен, поскольку контейнеры используют ядро хостовой ОС, а не отдельную операционную систему, что экономит ресурсы. Контейнеры быстро создаются и запускаются, ускоряя процессы разработки и развертывания. Кроме того, Docker легко масштабируется, позволяя добавлять новые контейнеры или использовать оркестраторы для управления многоконтейнерными приложениями. В контейнерах можно хранить все зависимости, что исключает проблемы с несовместимостью версий.

**Недостатки Docker**

Несмотря на множество преимуществ, Docker имеет и свои недостатки. Одним из них является ограниченная изоляция по сравнению с виртуальными машинами: контейнеры используют одно ядро ОС, что может привести к проблемам с безопасностью и конфиденциальностью при неправильной настройке. Docker также требует определенных знаний и опыта для эффективного использования, что может быть трудным для новичков. Кроме того, управление состоянием контейнеров в некоторых случаях может быть сложным, особенно при работе с приложениями, требующими постоянного хранения данных, поскольку контейнеры по умолчанию не сохраняют состояние. Docker также не идеально подходит для всех типов приложений — например, для тех, которые требуют высокой производительности или специфической конфигурации ОС.

## **1.4) Описание технологии PostgreSQL**

PostgreSQL — это объектно-реляционная система управления базами данных (СУБД), которая используется для хранения, обработки и управления данными. Она является одной из самых популярных и мощных открытых СУБД, известной своей надежностью, производительностью и поддержкой расширенных функций. PostgreSQL поддерживает стандарт SQL и обеспечивает возможность работы с данными как в табличном, так и в объектно-ориентированном формате.

Система поддерживает транзакции, обеспечивая высокую целостность данных, и имеет функции для работы с большими объемами данных, а также для обработки сложных запросов. PostgreSQL поддерживает репликацию данных, что позволяет использовать ее в распределенных системах, и может работать в кластерных конфигурациях для обеспечения высокой доступности и масштабируемости. Она также предоставляет богатый набор расширений и поддерживает различные типы данных, включая текст, числа, JSON, и даже геопространственные данные.

PostgreSQL широко используется в веб-приложениях, аналитике, корпоративных системах и многих других областях благодаря своей гибкости и высокой производительности.

**Преимущества PostgreSQL**

PostgreSQL обладает рядом преимуществ, которые делают его популярным выбором для разработки и использования баз данных. Во-первых, это открытость и бесплатность. PostgreSQL — это открытая система, что означает отсутствие лицензионных сборов и возможность модификации исходного кода для подгонки под специфические нужды. Во-вторых, надежность и целостность данных. PostgreSQL поддерживает транзакции и ACID-принципы, что обеспечивает высокий уровень защиты данных и их целостности даже при сбоях в системе.

В-третьих, PostgreSQL обладает расширяемостью. Он поддерживает создание собственных типов данных, функций, операторов и индексов, что позволяет адаптировать базу данных под специфические требования. Система поддерживает разнообразие типов данных, включая JSON, XML и геопространственные данные, что делает ее удобной для работы с различными видами информации.

Еще одно важное преимущество — это масштабируемость. PostgreSQL поддерживает различные методы репликации, включая синхронную и асинхронную репликацию, а также кластеризацию, что позволяет эффективно использовать его в больших распределенных системах. Он также предлагает хорошую производительность и возможности для оптимизации запросов через индексы и другие механизмы.

Кроме того, PostgreSQL предоставляет поддержку SQL и NoSQL возможностей, таких как работа с JSON, что позволяет гибко выбирать подход к хранению данных. Он поддерживает многопользовательскую работу с высокими требованиями к безопасности и аутентификации, что делает его идеальным для корпоративных приложений.

**Недостатки PostgreSQL**

Несмотря на множество преимуществ, PostgreSQL имеет и некоторые недостатки. Во-первых, сложность настройки и администрирования. В отличие от более простых систем управления базами данных, PostgreSQL требует более тщательной настройки и внимания со стороны администратора, особенно при работе с большими объемами данных и сложными конфигурациями.

Во-вторых, производительность при работе с очень большими объемами данных может быть ниже, чем у некоторых других СУБД, таких как MySQL или NoSQL решения, которые лучше подходят для специфичных рабочих нагрузок, например, для обработки больших данных в реальном времени.

Еще одним недостатком является ограниченная поддержка параллелизма. Несмотря на улучшения в последних версиях PostgreSQL, система все еще может испытывать трудности с эффективным распределением работы по нескольким процессорам или ядрам при выполнении сложных запросов в больших приложениях.

Также, встроенные средства для горизонтального масштабирования в PostgreSQL ограничены по сравнению с решениями, специально разработанными для распределенных систем, как, например, Cassandra или MongoDB. Это означает, что для масштабирования PostgreSQL на несколько узлов обычно требуется дополнительное ПО или решения третьих сторон.

Кроме того, время отклика при высоких нагрузках и потребление памяти могут быть проблемой в специфических случаях, таких как работа с большими базами данных с высоким числом одновременных соединений или сложных запросах.

## **1.5) Сравнительная характеристика Java**

Сравнивая Java с Python, можно отметить, что Python — это интерпретируемый язык с упором на простоту и читаемость кода. Python идеально подходит для быстрого прототипирования и проектов, где скорость разработки важнее производительности. Однако Python уступает Java по производительности, особенно в многопоточных приложениях. Java, благодаря своей виртуальной машине и оптимизациям, более эффективно управляет памятью и потоками, что делает его предпочтительным для высоконагруженных серверных приложений.

Если сравнить Java с C#, можно заметить, что оба языка имеют схожую синтаксическую структуру и оба активно используются для разработки корпоративных приложений. Тем не менее, C# более тесно интегрирован с экосистемой Microsoft и обычно используется в среде Windows, в то время как Java сохраняет нейтральность по отношению к платформам. Java также имеет более широкую поддержку со стороны сообщества и больше готовых библиотек и инструментов с открытым исходным кодом.

JavaScript, в отличие от Java, является языком, ориентированным на веб-разработку и взаимодействие с пользовательскими интерфейсами в браузере. Хотя на JavaScript можно создавать серверные приложения (например, с использованием Node.js), он не предоставляет той же степени строгости и производительности, что Java. Java лучше справляется с многопоточностью и обработкой сложной бизнес-логики.

Go (или Golang) — это относительно новый язык, созданный для упрощения разработки многопоточных приложений. Хотя он превосходит Java в плане простоты и скорости компиляции, Java остается лидером благодаря зрелой экосистеме, множеству проверенных временем библиотек и обширной документации. Go пока уступает Java в плане поддержки сложных корпоративных решений.

Ключевая причина использовать Java для разработки клиент-серверного приложения заключается в ее универсальности и надежности. Java предлагает мощные фреймворки, такие как Spring и Java EE, которые ускоряют разработку серверной части. Кроме того, благодаря JVM приложения на Java могут быть развернуты на любых серверах независимо от операционной системы. Поддержка многопоточности, высокая производительность, стабильность, развитая экосистема библиотек и сообщество разработчиков делают Java идеальным выбором для создания надежных и масштабируемых клиент-серверных приложений.

**1.6) Сравнительная характеристика Go**

Golang, в отличие другиз языков, например, Python, JavaScript и C#, представляет собой компилируемый язык, что обеспечивает ему высокую производительность, обладая выдающимися мощностями для многопоточных приложений. Python же является интерпретируемым языком, что делает его медленнее, но простота и читаемость делают его идеальным для быстрой разработки прототипов. JavaScript, работающий в браузере, обладает асинхронной природой, что позволяет создавать интерактивные приложения, но производительность может варьироваться в зависимости от движка. C# работает на платформе .NET и показывает достойные показатели, хотя в некоторых задачах может уступать Golang, особенно в облачных архитектурах.

Что касается синтаксиса, Golang выделяется своей простотой и лаконичностью, но может быть менее гибким, чем динамичные языки. Python славится своей читаемостью, идеально подходящей для новичков. JavaScript имеет сложную структуру, что может усложнить его изучение, в то время как C# требует больше времени на освоение из-за обширного набора возможностей.

Golang поддерживает процессное программирование и горутины, что делает его подходящим для высокопроизводительных серверов. Python поддерживает множество парадигм, включая объектно-ориентированное программирование. JavaScript ориентирован на веб-разработку, а C# поддерживает различные парадигмы, делая его универсальным.

Экосистема Golang стремительно растет, однако число доступных библиотек меньше, чем у Python или JavaScript. Python обладает огромной экосистемой библиотек, особенно в áreas науки о данных и веб-разработки. JavaScript, как популярный язык для веб-разработки, предлагает множество библиотек и фреймворков. C# также имеет развитую экосистему, особенно для создания Windows-приложений.

Наконец, сообщество Golang активно растет, но по-прежнему меньше, чем у Python или JavaScript. Python имеет одно из крупнейших сообществ и множество учебных ресурсов, JavaScript обладает активной поддержкой, а C# пользуется поддержкой от Microsoft, что обеспечивает регулярное обновление и улучшение языка. Эти характеристики помогут оценить, в каких ситуациях стоит выбирать Golang по сравнению с другими языками программирования.

## **1.7) Характеристики Java**

Java — это универсальный, объектно-ориентированный язык программирования, созданный для написания надежных, масштабируемых и переносимых приложений. Одной из ключевых характеристик Java является его кросс-платформенность, которая достигается благодаря использованию виртуальной машины Java (JVM). Приложения на Java компилируются в байт-код, который может исполняться на любой системе с установленной JVM, независимо от операционной системы. Это делает Java особенно удобным для разработки клиент-серверных приложений, которые должны работать в разнообразных вычислительных средах.

Java обладает строгой типизацией, что помогает разработчикам избежать множества ошибок на этапе компиляции. Это особенно важно для сложных систем, таких как клиент-серверные приложения, где требуется высокая надежность. В дополнение к этому, встроенная система управления памятью с автоматическим сборщиком мусора (Garbage Collector) снижает вероятность утечек памяти и делает управление ресурсами более эффективным.

Одним из основных преимуществ Java является богатая экосистема библиотек и фреймворков, которые упрощают разработку клиент-серверных приложений. Например, фреймворк Spring предоставляет мощные инструменты для построения RESTful веб-сервисов, обработки запросов и управления бизнес-логикой. Java EE (Jakarta EE) предоставляет стандартизированный набор инструментов для создания корпоративных приложений, включая поддержку транзакций, управления потоками и интеграции с базами данных. Эти фреймворки и инструменты позволяют разработчикам сосредоточиться на бизнес-логике, оставляя низкоуровневую обработку инфраструктуры фреймворкам.

Поддержка многопоточности — еще одна сильная сторона Java. Клиент-серверные приложения часто обрабатывают множество запросов от различных пользователей одновременно, что требует эффективного управления потоками. Java предоставляет встроенные механизмы для работы с потоками, включая библиотеку java.util.concurrent, которая упрощает управление задачами, потоками и синхронизацией. Благодаря этому Java идеально подходит для создания высоконагруженных серверов, способных обрабатывать десятки тысяч запросов в секунду.

Безопасность также играет ключевую роль в Java, что делает язык предпочтительным для клиент-серверных приложений, где важно защитить данные пользователей. Java предоставляет встроенные инструменты для шифрования, проверки подлинности и авторизации. Кроме того, JVM работает в изолированной среде, что снижает вероятность выполнения вредоносного кода.

Преимущество Java в стабильности и масштабируемости подтверждается многолетним использованием языка в корпоративной среде. Язык развивается уже более двух десятилетий и поддерживается крупным сообществом разработчиков. Это означает, что разработчики имеют доступ к обширной документации, обучающим материалам и готовым решениям, что ускоряет процесс разработки.

Клиент-серверные приложения требуют высокой производительности, а Java предоставляет средства для ее оптимизации. Современные версии JVM содержат Just-In-Time (JIT) компилятор, который динамически оптимизирует код во время выполнения, улучшая производительность. Кроме того, Java поддерживает интеграцию с такими технологиями, как базы данных (через JDBC, JPA или Hibernate), что упрощает управление данными.

Таким образом, Java является идеальным выбором для разработки клиент-серверных приложений благодаря своей кросс-платформенности, производительности, богатой экосистеме инструментов, встроенной поддержке многопоточности и безопасности. Этот язык обеспечивает разработчикам стабильность, удобство и мощь, необходимые для создания масштабируемых и надежных решений, удовлетворяющих требованиям современного бизнеса.

**1.8) Характеристики Go**

Go — это современный, компилируемый язык программирования, разработанный для создания высокопроизводительных, надежных и эффективных приложений. Одной из отличительных черт Golang является его простота и лаконичность, которые помогают разработчикам сосредоточиться на решении задач, не отвлекаясь на сложные синтаксические конструкции. Благодаря этому язык находит широкий отклик в разработке облачных и серверных приложений.

Одной из ключевых характеристик Golang является его поддержка параллелизма, что делает его идеальным выбором для разработки многопоточных и сетевых приложений. Язык предоставляет уникальную конструкцию под названием "горутины", которые позволяют создавать легковесные потоки, поддерживающие плавное выполнение, экономя при этом ресурсы. Кроме того, использование каналов (channels) для коммуникации между горутинами упрощает управление синхронизацией и обменом данными.

Golang был создан с акцентом на производительность. Компилятор Go генерирует быстрый машинный код, что обеспечивает высокую скорость выполнения приложений. Язык имеет встроенные оптимизации, такие как статическая типизация и современный механизм управления памятью, который минимизирует вероятность утечек и недостатков, связанных с управлением ресурсами. Также Golang характеризуется Time-to-Compile, что имеется в виду время компиляции, которое ассоциируется с высокой производительностью разработки.

Кроме того, Golang имеет богатую стандартную библиотеку, которая облегчает разработку. Встроенные пакеты для сетевых операций, работы с потоками и обработки данных делают Go универсальным инструментом для создания различных типов приложений. Например, пакет "net/http" предоставляет мощные функции для построения веб-серверов и RESTful API, что упрощает разработку высоконагруженных систем.

Безопасность также является важным аспектом языка. Golang включает механизмы для обработки ошибок, работающие на уровне языка, что значительно упрощает отладку и тестирование программ. Простая модель обработки ошибок помогает избежать ненадежного кода и повышает устойчивость приложений.

Golang набирает популярность благодаря активному сообществу и поддержке крупных организаций, использующих его в своих проектах. Это гарантирует рост и развитие языка, а также доступность широкого спектра библиотек и фреймворков, таких как Gin и Echo, которые помогают ускорять процесс разработки.

Таким образом, Golang является идеальным выбором для создания высокопроизводительных, надежных и масштабируемых серверных приложений. Его простота, поддержка параллелизма, богатая экосистема инструментов и высокая производительность делают его привлекательным для разработчиков, стремящихся создавать эффективные решения, отвечающие требованиям современного бизнеса.

## **1.9) Клиент-серверная архитектура**

Клиент-серверная архитектура — это модель, в которой приложения разделены на две основные части: клиент и сервер. Клиент — это приложение или устройство, которое отправляет запросы к серверу, а сервер — это система, которая обрабатывает эти запросы и предоставляет ответы. В этой архитектуре сервер отвечает за обработку данных, хранение информации и выполнение логики, в то время как клиент отвечает за взаимодействие с пользователем, представление данных и отправку запросов на сервер.

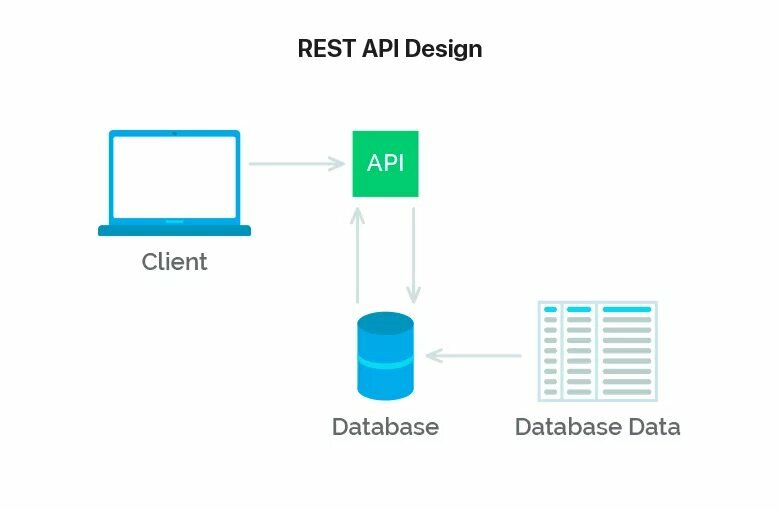


Рисунок 3. Структура Клиент серверной архитектуры

Обычно сервер и клиент работают на разных устройствах, но могут находиться и на одном сервере. Серверы могут быть централизованными, например, в дата-центрах, или распределенными, когда несколько серверов работают вместе, чтобы обеспечить высокую доступность и производительность. Серверы обычно обрабатывают множество запросов от разных клиентов одновременно, а также могут работать с базами данных и другими системами для выполнения операций.

Клиент-серверная архитектура позволяет эффективно распределять нагрузку и ресурсы между сервером и клиентом. Она обеспечивает гибкость, так как изменение логики работы на сервере не требует изменений на стороне клиента. Это также позволяет централизованно управлять данными, улучшая безопасность, поддержку и масштабируемость. Этот подход используется в большинстве современных веб-приложений, где браузер или мобильное приложение (клиент) взаимодействует с сервером через сеть для получения данных или выполнения операций.

# **2. РАЗРАБОТКА ПРИЛОЖЕНИЯ**

## 

## **2.1) Структура проекта**

При разработке проекта была выбрана следующая структура проекта (Рис. 4)

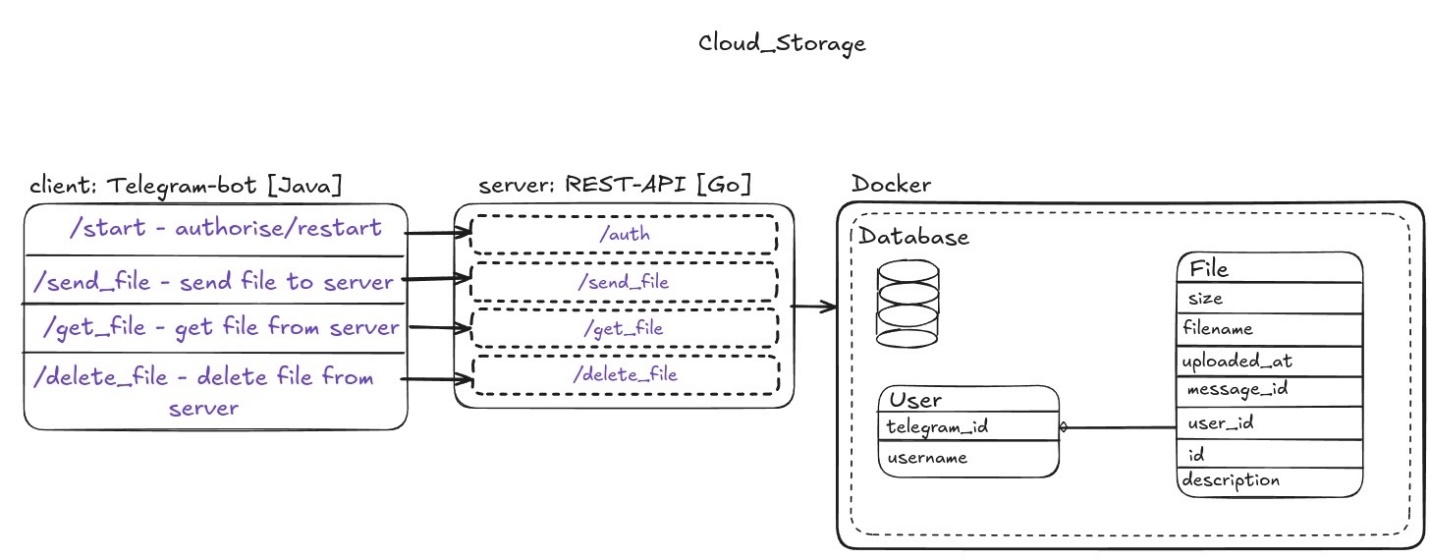


Рис. 4. Структура проекта

Общение пользователя с сервисом происходит за счет сообщений телеграм боту, каждое сообщение принимается на клиенте, обрабатывается и на основе содержимого сообщения отправляется соответствующий запрос на сервер, интерфейс которого представляет собой REST-api. Запрос, полученный на сервере в зависимости от адреса и метода обрабатывается соответствующим обработчиком. Обработчик передает данные сервису, который обрабатывает данные и отправляет запрос в базу данных. Далее, на основе вернувшихся из БД данных формируется ответ клиенту. Клиент приняв ответ выводит соответствующее сообщение пользователю о результате операции.

**Полное описание процесса старта работы приложения**

Первым делом требуется запустить Docker контейнер с базой данных. Далее запустить сервер, который подключается к контейнеру. Далее запустить клиент телеграм бота.

**Миграции в базу данных**

Миграции в базу данных описаны в файлах в директории migrations. Они применяются отдельной утилитой golang-migrate и подготавливают структуру базы данных до требований проекта.

**Запуск сервера**

Сервер запускается из единственного ендпоинта, который представляет собой go файл. Процессу присваивается порт 8080, на который потом будут приходить обращения.

**Запуск телеграм бота**

Телеграм бот запускается с заранее полученным токеном бота, чтобы получить доступ к управлению ботом.

## **2.2) Разработка серверной части**

Сервер может обрабатывает следующие HTTP-запросы:

**POST /auth**

**Body**: {

telegram\_id int,

username string

}

Сервер добавляет пользователя в базу данных и возвращает ответ 200, если все прошло успешно, при возникновении ошибок возвращает 400.

*func (r Repository) Create(ctx context.Context, entity \*model.Users) error {*

*query := `INSERT INTO Users (telegram\_id, username)*

*VALUES (:telegram\_id, :username)*

*ON CONFLICT (telegram\_id)*

*DO UPDATE SET username = EXCLUDED.username;`*

*\_, err := r.Db.NamedQueryContext(ctx, query, entity)*

*if err != nil {*

*return db.HandleError(err)*

*}*

*return db.HandleError(err)*

*}*

**GET /get\_file/{id}**

В теле передается id файла в базе данных, по этому id возвращаются данные о файле из базы данных

*{*

*id int,*

*filename string,*

*file\_size int,*

*uploaded\_at timestamp,*

*file\_body string,*

*}*

*func (r Repository) FindFile(ctx context.Context, id int) (model.Files, error) {*

*entity := model.Files{}*

*query := "SELECT id, description, file\_size, filename, uploaded\_at, user\_id, message\_id, file\_body FROM Files WHERE id = $1;"*

*row := r.Db.QueryRowContext(ctx, query, id)*

*err := row.Scan(*

*&entity.Id,*

*&entity.Description,*

*&entity.File\_size,*

*&entity.Filename,*

*&entity.Uploaded\_at,*

*&entity.User\_id,*

*&entity.Message\_id,*

*&entity.File\_body,*

*)*

*if err == sql.ErrNoRows {*

*err = nil*

*}*

*return entity, db.HandleError(err)*

*}*

**GET /delete\_file/{id}**

Удаляет файл с выбранным id из базы данных, возвращает статус 200, если все прошло успешно, иначе возвращает 400.

*func (r Repository) DeleteFile(ctx context.Context, id int) error {*

*query := "DELETE FROM Files WHERE id = $1;"*

*\_, err := r.Db.ExecContext(ctx, query, id)*

*if err != nil {*

*return err*

*}*

*return nil*

*}*

## 

**GET /get\_files/{user\_id}**

Возвращает список файлов пользователя, user\_id которого передается в запросе

Возвращает JSON такого формата

{

“files”:

[

{

“id”: <int>,

“filename”: <string>

},

{

“id”: <int>,

“filename”: <string>

},

…

]

}

*func (r Repository) ListFiles(ctx context.Context, user\_id int, page int, offset int) ([]FileHead, error) {*

*var entities []FileHead*

*query := "SELECT id, filename FROM Files WHERE user\_id = $1 LIMIT $2 OFFSET $3;"*

*err := r.Db.SelectContext(ctx, &entities, query, user\_id, offset, offset\*page)*

*return entities, db.HandleError(err)*

*}*

***POST /send\_file***

На сервер передаются все сведения о файле и его содержимое в кодировке base64, они записываются в базу данных и используются далее.

{

Description string

File\_size int

Filename string

Uploaded\_at time.Time

User\_id int

Message\_id int

File\_body string

}

*func (s Service) CreateFile(ctx context.Context, params CreateFileParams) error {*

*if \_, err := govalidator.ValidateStruct(params); err != nil {*

*return erru.ErrArgument{Wrapped: err}*

*}*

*tx, err := s.repo.Db.BeginTxx(ctx, nil)*

*if err != nil {*

*return err*

*}*

*// Defer a rollback in case anything fails.*

*defer tx.Rollback()*

*entity := model.Files{*

*Description: params.Description,*

*File\_size: params.File\_size,*

*Filename: params.Filename,*

*Uploaded\_at: params.Uploaded\_at,*

*User\_id: params.User\_id,*

*Message\_id: params.Message\_id,*

*File\_body: params.File\_body,*

*}*

*\_, err = s.repo.CreateFile(ctx, entity)*

*if err != nil {*

*fmt.Println(err, "qwe")*

*return err*

*}*

*err = tx.Commit()*

*return err*

*}*

­­­­­­­­­­­­­­­­­­­­­­­­­­­**Вспомогательные модули**

В проекте используются следующие библиотеки:

mux — это библиотека, которая предоставляет маршрутизатор для обработки HTTP-запросов в приложениях на Go. Она позволяет сопоставлять запросы с обработчиками, используя пути с параметрами, такими как идентификаторы (`/users/{id}`), и учитывать методы запросов, например, GET или POST. Это упрощает создание и управление сложными API с большим количеством маршрутов.

logrus — это мощный инструмент для логирования в Go. Он позволяет записывать структурированные логи, добавляя к ним дополнительную информацию, такую как уровень важности (Debug, Info, Warn, Error) или контекст выполнения. Логи могут быть записаны в удобном формате, например, JSON, что упрощает их анализ и интеграцию с системами мониторинга.

sqlx — это расширение стандартной библиотеки `database/sql` в Go, которое добавляет удобные функции для работы с базами данных. Оно поддерживает автоматическое сопоставление данных из запросов с полями структур, выполнение SQL-запросов с упрощенной обработкой параметров и чтение сложных результатов. Это делает работу с базами данных более простой и эффективной.

pq — это драйвер для PostgreSQL, написанный на Go. Он работает с библиотекой database/sql, предоставляя все необходимые функции для подключения к PostgreSQL, выполнения запросов, работы с транзакциями и других операций.

govalidator — это библиотека для проверки и очистки данных. Она содержит множество готовых функций для валидации строк (например, проверки электронной почты, URL или IP-адресов), структур и коллекций. Используется для проверки входных данных в приложении.

godotenv — это библиотека, которая помогает загружать переменные окружения из файла .env . Это удобно для работы с конфигурацией приложений, так как позволяет хранить параметры, такие как пароли или адреса баз данных, отдельно от исходного кода.

envconfig — это библиотека для управления конфигурацией через переменные окружения. Она позволяет удобно связывать переменные окружения с полями структур в Go, поддерживает настройку параметров по умолчанию и автоматическое приведение типов. Это упрощает управление настройками приложений.

**/pkg/db/db.go**

Вспомогательный код хранится в директории pkg. Этот код может быть рассмотрен в качестве библиотеки и может использоваться в других проектах.

В директории pkg/db хранится код для соединения и взаимодействия с базой данных, из файла конфигурации берется вся ифнормация для подключения: пароль, имя пользователя, названия базы данных, адрес и порт на которой БД запущена

*package db*

*import (*

*"fmt"*

*"github.com/jmoiron/sqlx"*

*\_ "github.com/lib/pq"*

*)*

*type ConfingDB struct {*

*Host string*

*Port int*

*User string*

*Password string*

*Name string*

*}*

*func Connect(cnf ConfingDB) (\*sqlx.DB, error) {*

*dsn := fmt.Sprintf(*

*"host=%s port=%d user=%s password=%s dbname=%s sslmode=disable",*

*cnf.Host,*

*cnf.Port,*

*cnf.User,*

*cnf.Password,*

*cnf.Name,*

*)*

*db, err := sqlx.Connect("postgres", dsn)*

*return db, err*

*}*

**main.go**

Код запускает REST API сервер. В функции main вызывается функция run, в которую передается контекст context.Background(). Если при выполнении функции run возникает ошибка, она логируется с завершением работы программы. В функции run создается сервер с помощью rest.NewServer(). Если при создании сервера возникает ошибка, она возвращается. Затем сервер запускается методом server.Run(ctx), куда передается контекст выполнения. Если при запуске возникает ошибка, она также возвращается.

*package main*

*import (*

*"context"*

*"github.com/fir1/rest-api/http/rest"*

*"log"*

*)*

*func main() {*

*if err := run(context.Background()); err != nil {*

*log.Fatalf("%+v", err)*

*}*

*}*

*func run(ctx context.Context) error {*

*server, err := rest.NewServer()*

*if err != nil {*

*return err*

*}*

*err = server.Run(ctx)*

*return err*

*}*

**auth.go**

Обработчик аутентификации пользователя, который принимает telegram\_id и username

*func (s service) Auth() http.HandlerFunc {*

*type request struct {*

*Telegram\_id int `json:"telegram\_id"`*

*Username string `json:"username"`*

*}*

*type response struct {*

*Message string `json:"message"`*

*}*

*return func(w http.ResponseWriter, r \*http.Request) {*

*req := request{}*

*// Try to decode the request body into the struct. If there is an error,*

*// respond to the client with the error message and a 400 status code.*

*err := s.decode(r, &req)*

*if err != nil {*

*fmt.Println(err)*

*s.respond(w, err, 0)*

*return*

*}*

*err = s.cloudStorageService.CreateUser(r.Context(), cloudStorageService.CreateParams{*

*Telegram\_id: req.Telegram\_id,*

*Username: req.Username,*

*})*

*if err != nil {*

*s.respond(w, err, 0)*

*return*

*}*

*s.respond(w, response{Message: "OK"}, http.StatusOK)*

*}*

*}*

**delete.go**

Обработчик, отвечающий за удаление файла, принимающий в запросе id файла

*func (s service) Delete() http.HandlerFunc {*

*return func(w http.ResponseWriter, r \*http.Request) {*

*vars := mux.Vars(r)*

*id, err := strconv.Atoi(vars["id"])*

*if err != nil {*

*s.respond(w, err, 0)*

*return*

*}*

*err = s.cloudStorageService.DeleteFile(r.Context(), id)*

*if err != nil {*

*s.respond(w, err, 0)*

*return*

*}*

*s.respond(w, nil, http.StatusOK)*

*}*

*}*

**files\_list.go**

Обработчик, отвечающий за вывод списка всех файлов пользователя, принимает в запросе парметры page, offset, id

*const OFFSET = 10*

*func (s service) FilesList() http.HandlerFunc {*

*type response struct {*

*Files []repository.FileHead `json:"files"`*

*}*

*return func(w http.ResponseWriter, r \*http.Request) {*

*vars := mux.Vars(r)*

*// Try to decode the request body into the struct. If there is an error,*

*// respond to the client with the error message and a 400 status code.*

*var page, offset int*

*page = 0*

*if param := r.URL.Query().Get("page"); len(param) > 0 {*

*page, \_ = strconv.Atoi(param)*

*}*

*offset = OFFSET*

*if param := r.URL.Query().Get("offset"); len(param) > 0 {*

*offset, \_ = strconv.Atoi(param)*

*}*

*user\_id, err := strconv.Atoi(vars["user\_id"])*

*if err != nil {*

*s.respond(w, err, 0)*

*return*

*}*

*filesListResponse, err := s.cloudStorageService.GetFiles(r.Context(), cloudStorageService.GetFilesParams{*

*User\_id: user\_id,*

*Page: page,*

*Offset: offset,*

*})*

*if err != nil {*

*s.respond(w, err, 0)*

*return*

*}*

*s.respond(w, response{Files: filesListResponse}, http.StatusOK)*

*}*

*}*

**get.go**

Обработчик, отвечающий за получение одного фала, принимает в запросе id файла

*func (s service) GetFile() http.HandlerFunc {*

*type response struct {*

*Id int `json:"id"`*

*Filename string `json:"filename"`*

*File\_size int `json:"file\_size"`*

*Uploaded\_at time.Time `json:"uploaded\_at"`*

*File\_body string `json:"file\_body"`*

*}*

*return func(w http.ResponseWriter, r \*http.Request) {*

*vars := mux.Vars(r)*

*id, \_ := strconv.Atoi(vars["id"])*

*getFileResponse, err := s.cloudStorageService.GetFile(r.Context(), id)*

*if err != nil {*

*s.respond(w, err, 0)*

*return*

*}*

*s.respond(w, response{*

*Id: getFileResponse.Id,*

*Filename: getFileResponse.Filename,*

*File\_size: getFileResponse.File\_size,*

*Uploaded\_at: getFileResponse.Uploaded\_at,*

*File\_body: getFileResponse.File\_body,*

*}, http.StatusOK)*

*}*

*}*

**middleware\_logger.go**

В данном модуле запрос проходит предвариетльную обработку, в нем происходит логирование и задаются методы класса для работы с ответом и запросом. Код реализует middleware для логирования запросов и ответов в HTTP-сервере. Основная часть функционала реализована через обертку responseWriter, которая расширяет стандартный интерфейс http.ResponseWriter. Она позволяет сохранять статус ответа и его тело для последующего логирования. В обертке записываются код ответа и тело, а также проверяется, записывались ли заголовки или тело, чтобы избежать повторной записи.

Функция MiddlewareLogger добавляет логику логирования к обработчику запросов. Она проверяет путь запроса и пропускает его без логирования, если это запрос на /healthz. Это сделано для исключения внутренних проверок здоровья системы из логов. Если во время обработки запроса возникает ошибка, она перехватывается, и сервер возвращает статус 500. Тело запроса читается и восстанавливается для логирования. Далее замеряется время выполнения обработки запроса. Ответ оборачивается в responseWriter, что позволяет сохранить информацию о статусе и теле ответа. После обработки запроса логируются данные о пути, методе, теле запроса, статусе ответа, теле ответа и времени выполнения.

*// responseWriter is a minimal wrapper for http.ResponseWriter that allows the*

*// written HTTP status code to be captured for logging. This type will implement http.ResponseWriter.*

*type responseWriter struct {*

*http.ResponseWriter*

*status int*

*body []byte*

*wroteHeader bool*

*wroteBody bool*

*}*

*func wrapResponseWriter(w http.ResponseWriter) \*responseWriter {*

*return &responseWriter{ResponseWriter: w}*

*}*

*func (rw \*responseWriter) Status() int {*

*return rw.status*

*}*

*func (rw \*responseWriter) WriteHeader(code int) {*

*if rw.wroteBody {*

*return*

*}*

*rw.status = code*

*rw.ResponseWriter.WriteHeader(code)*

*rw.wroteHeader = true*

*}*

*func (rw \*responseWriter) Write(body []byte) (int, error) {*

*if rw.wroteBody {*

*return 0, nil*

*}*

*i, err := rw.ResponseWriter.Write(body)*

*if err != nil {*

*return 0, err*

*}*

*rw.body = body*

*return i, err*

*}*

*func (rw \*responseWriter) Body() []byte {*

*return rw.body*

*}*

*// middlewarMiddlewareLoggereLogger logs the incoming HTTP request and response. Enable it only for debug purpose disable it on production.*

*func (s service) MiddlewareLogger() func(http.Handler) http.Handler {*

*return func(next http.Handler) http.Handler {*

*fn := func(w http.ResponseWriter, r \*http.Request) {*

*if r.URL.Path == "/healthz" {*

*// Call the next handler don't log if it is internal request from health check of Kubernetes*

*next.ServeHTTP(w, r)*

*return*

*}*

*defer func() {*

*if err := recover(); err != nil {*

*w.WriteHeader(http.StatusInternalServerError)*

*}*

*}()*

*requestBody, err := s.readRequestBody(r)*

*if err != nil {*

*s.respond(w, err, 0)*

*return*

*}*

*s.restoreRequestBody(r, requestBody)*

*logMessage := fmt.Sprintf("path:%s, method: %s, requestBody: %v", r.URL.EscapedPath(), r.Method, string(requestBody))*

*start := time.Now()*

*wrapped := wrapResponseWriter(w)*

*next.ServeHTTP(wrapped, r)*

*logMessage = fmt.Sprintf("%s, responseStatus: %d, responseBody: %s", logMessage, wrapped.Status(), string(wrapped.Body()))*

*s.logger.Infof("%s, duration: %v", logMessage, time.Since(start))*

*}*

*return http.HandlerFunc(fn)*

*}*

*}*

**routes.go**

Код настраивает маршруты (endpoints) HTTP-сервера, регистрирует обработчики запросов и добавляет middleware для логирования. Функция Register принимает три аргумента: маршрутизатор mux.Router для управления путями запросов, логгер logrus.Logger для логирования событий и базу данных sqlx.DB для взаимодействия с данными.

Сначала создается новый экземпляр обработчика запросов с помощью функции newHandler, которой передаются логгер и база данных. Затем к маршрутизатору добавляется middleware MiddlewareLogger, который логирует все запросы и ответы. После этого настраиваются маршруты. Для каждого пути указывается обработчик, который будет вызван при обращении к этому пути, а также метод HTTP-запроса, который поддерживает данный обработчик.

Обработчики выполняют следующие действия:

* /healthz — возвращает статус здоровья системы. Этот путь обычно используется для проверки состояния сервера.
* /v1/api/auth — обрабатывает POST-запросы для аутентификации пользователя.
* /v1/api/get\_file/{id} — обрабатывает GET-запросы для получения файла по его идентификатору.
* /v1/api/delete\_file/{id} — обрабатывает DELETE-запросы для удаления файла по его идентификатору.
* /v1/api/get\_files/{user\_id} — обрабатывает GET-запросы для получения списка файлов пользователя по его идентификатору.
* /v1/api/send\_file — обрабатывает POST-запросы для отправки файла.

Каждый маршрут связывается с соответствующей функцией обработчика, которая реализует нужную логику обработки запроса.

*package handlers*

*import (*

*"net/http"*

*"github.com/gorilla/mux"*

*"github.com/jmoiron/sqlx"*

*"github.com/sirupsen/logrus"*

*)*

*func Register(r \*mux.Router, lg \*logrus.Logger, db \*sqlx.DB) {*

*handler := newHandler(lg, db)*

*// adding logger middleware*

*r.Use(handler.MiddlewareLogger())*

*r.HandleFunc("/healthz", handler.Health())*

*r.HandleFunc("/v1/api/auth", handler.Auth()).Methods(http.MethodPost)*

*r.HandleFunc("/v1/api/get\_file/{id}", handler.GetFile()).Methods(http.MethodGet)*

*r.HandleFunc("/v1/api/delete\_file/{id}", handler.Delete()).Methods(http.MethodDelete)*

*r.HandleFunc("/v1/api/get\_files/{user\_id}", handler.FilesList()).Methods(http.MethodGet)*

*r.HandleFunc("/v1/api/send\_file", handler.SendFile()).Methods(http.MethodPost)*

*}*

***/.env***

Файл конфигурации содержит следующие поля:

1. DATABASE\_HOST — адрес хоста базы данных. В данном случае это localhost, что указывает на то, что база данных работает на том же сервере, где выполняется приложение.
2. DATABASE\_PORT — порт, на котором работает база данных. Здесь указан порт 5433.
3. DATABASE\_USER — имя пользователя для подключения к базе данных. В данном примере пользователь называется myuser.
4. DATABASE\_PASSWORD — пароль для указанного пользователя базы данных. Здесь пароль — mypassword.
5. DATABASE\_NAME — имя базы данных, к которой подключается приложение. В данном случае это mydatabase.
6. SERVER\_PORT — порт, на котором запускается сервер приложения. В этом примере сервер работает на порту 8080.

Этот файл конфигурации задает параметры для подключения к базе данных PostgreSQL и настройки веб-сервера.

**models.go**

В этом файле описаны модели сущностей базы данных, соответствующие таблицам. Модель `Users` представляет таблицу пользователей, где хранятся данные о пользователях Telegram. У нее есть два поля: `Telegram\_id`, которое служит уникальным идентификатором пользователя в Telegram, и `Username`, которое хранит имя пользователя.

Модель `Files` описывает таблицу с информацией о загруженных файлах. Она включает следующие поля: `Id`, которое является уникальным идентификатором файла; `Description`, содержащее описание файла; `File\_size`, где хранится размер файла в байтах; `Filename`, содержащее имя файла; `Uploaded\_at`, указывающее время загрузки файла; `User\_id`, которое связывает файл с пользователем из таблицы `Users`; `Message\_id`, связанный с сообщением, к которому относится файл; и `File\_body`, где хранится тело файла в виде строки. Эти модели используются для работы с базой данных, где каждая структура представляет таблицу, а ее поля — столбцы таблицы.

*package model*

*import "time"*

*type Users struct {*

*Telegram\_id int `db:"telegram\_id"`*

*Username string `db:"username"`*

*}*

*type Files struct {*

*Id int `db:"id"`*

*Description string `db:"description"`*

*File\_size int `db:"file\_size"`*

*Filename string `db:"filename"`*

*Uploaded\_at time.Time `db:"uploaded\_at"`*

*User\_id int `db:"user\_id"`*

*Message\_id int `db:"message\_id"`*

*File\_body string `db:"file\_body"`*

*}*

## **2.3) Разаработка клиентской части**

**Общая информация**

Для обращения к серверной части приложения и базе данных используется мессенджер Telegram и обработчик событий (далее ‒ клиент). Пользователь имеет возможность манипулировать данными приложения через чат Телеграм-бота, используя фиксированный набор операций (см. табл. 1). Для удобства операции нумеруются целым числом от 0 до 4.

|  |  |  |
| --- | --- | --- |
| **Операции и их назначение** | | |
| **0** | **/start** | Аутентификация пользователя. Сохранение информации о пользователе в базе данных |
| **1** | **/send\_file** | Сохранение файла на сервере |
| **2** | **/get\_file** | Получение файла от сервера |
| **3** | **/delete\_file** | Удаление файла из базы данных |
| **4** | **/help** | Получение списка доступных команд |

Таблица 1. Операции над ботом

Для активации бота необходимо верно указать его имя (@dagprojbot) и токен. По ним клиент получает доступ к боту на сервере Telegram.

Исходный код клиента написан на языке **Java** при использовании библиотек **org.telegram.telegrambots:6.9.7.1** и **org.json.json:20250107**.

**Архитектура клиента**

Исходный код для Телеграмм-бота состоит из двух частей: ***boot***, ***api***. Первая часть занимается обработкой событий, которые возникает в чате бота мессенджера (далее ‒ чате). Каждому происходящему событию ставится в соответствие специальный метод в коде клиента, который определенным образом обрабатывает его. Вторая часть описывает сетевое взаимодействие с сервером, который принимает запросы пользователей. В ней происходит формирование HTTP-запросов, обработка приходящих HTTP-ответов, обработка исключений в случае ошибок в ходе передачи информации. Первая часть по необходимости обращается к второй для получения необходимых данных от сервера и базы данных.

В таблице 2 приведена структура файловой системы клиента.

|  |  |
| --- | --- |
| /src/main/java/ru.ikvt/dagprojbot | |
| /boot | /api |
| **CallbackResponseAction** | **SimpleZakharovAPI** |
| **DagaevProjectTelegramBot** | **ZakharovAPI** |
| **Main** |  |

Таблица 2. Структура файловой системы клиента

**/boot/DagaevProjectTelegramBot**

Это основной класс, который занимается обработкой всех событий, происходящих в чате. В качестве полей приведены имя бота, его токен, а также специальный объект, в котором происходит реализация сетевого взаимодействия для обращения к серверу. О нем речь пойдет позже.

Каждое событие, происходящее в чате, находит специальное отображение в Java-коде ‒ объекте **Update**. Объект содержит все необходимые данные о событиях, которые случились перед отправкой сервером Telegram указанного объекта, например, информация по сообщении, его содержимом, наличии/отсутствии в нем файлов, мультимедийных данных, редактировании сообщений и т. д. Объект **Update** поступает в качестве параметра в метод **onUpdateReceived** (см. сигнатуру в табл. 3), который вызывается каждый раз при возникновении нового события в чате. Отсюда следует, что, если выполнение операции требует наступления нескольких, в общем случае разных, событий, исходный код операции должен быть распределен между разными обработчиками событий. Перед наступлением нового события предыдущее должно быть полностью обработано. Обработка события завершается, когда выполнен исходный код, указанный в методе **onUpdateReceived**. Пока этого не случилось, следующее событие не может быть обработано.

|  |  |  |  |
| --- | --- | --- | --- |
| public | void | onUpdateReceived | (Update update) |

Таблица 3. Сигнатура метода onUpdateReceived

Клиент обрабатывает два основных события: отправки нового сообщения пользователем, нажатие на кнопку пользователем. Эти события находят отражение в двух соответствующих методах-обработчиках: **onMessageReceived** и **OnButtonClicked**. Любое взаимодействие пользователя с ботом начинается с отправки им сообщения незавимимо от того, содержит оно текстовую информацию или нет. Для выполнения операций аутентификации (**0**), получения файла (**2**), удаления файла (**3**) и вывода на экран списка доступных операций (**4**) требуется отправка сообщения с текстом, содержащим наименование операции, приведенное в скобках. Для выполнения операции отправки файла на сервер (**1**) текст сообщения не требуется, пользователю нужно просто прикрепить сохраняемый файл. Операции 0, 1, 4 требуют обработки только одного события ‒ отправки нового сообщения (**onMessageReceived**). Оставшиеся операции 2 и 3 требуют обработки уже двух разных событий (**onMessageReceived**, **OnButtonClicked**). По этой причине исходный код для реализации операций 0, 1 и 4 размещен только в методе-обработчике **onMessageReceived**, а для операций 2 и 3 ‒ в обоих. События отправки нового сообщения и нажатия на кнопку не являются независимыми: второе не наступит, пока первое не создаст ситуацию для его возникновения.

Помимо методов-обработчиков событий данный класс включает в себя и методы, логически разделяющие исходный код для реализации той или иной операции. Они включены обработчики **onMessageReceived** и **OnButtonClicked**.

Метод **auth**, исходя из названия, реализует операцию аутентификации пользователя. Внтури нее происходит удаленный вызов процедур (RPC) сервера посредством HTTP-соединения для сохранения данных о пользователе в базе данных.

Метод **send** реализует операцию отправки сообщения. Внтури нее происходит удаленный вызов RPC сервера посредством HTTP-соединения для сохранения файлов в базе данных.

Метод **getBase64str** извлекает из сервера Telegram файл, который отправил пользователь бота и преобразует его в строку с кодировкой base64. Все файлы хранятся в базе данных в виде строки с данной кодировкой. Извлеченный файл представлен в качестве объекта **GetFile**.

Метод **preprocessKeyboard** участвует в реализации операций 2 и 3. Он занимается подготовкой необходимых данных для вывода клавиатуры (**InlineKeyboardMarkup**) в чат пользователю. Нажимая на кнопки (**InlineKeyboardButton**) клавиатуры, пользователь выбирает, какие файлы он хочет получить/удалить. Каждая кнопка содержит название соответствующего ей файла. Для получения названий файлов необходимо обратиться к базе данных через RPC.

Метод **setButtons** непосредственно занимается выводом клавиатуры в чат пользователя. Как и предыдущий, принимает участие в реализации операций 2 и 3.

Нажатие пользователя на кнопку является отдельным событием, которое при помощи функций обратного вызова (**callback**) переносится клиенту в виде обновления. Данные, извлеченные из **callback**, используются для обработки очередного события в рамках выполнения операции. Потому следующие методы вызываются уже в рамках обработчика **onButtonClicked**.

Метод **respondCallback** реагирует на обратный вызов и обрабатывает данные, которые он принес в код. Там же принимается решение о том, какая операция должна быть реализована: получение файла или его удаление. Эта информация содержится в **callback**. При выполнении операции удаления происходит всего один RPC, который указывает базе данных удалить указанные файлы, на чем и завершается выполнение операции 3. Для выполнения операции получения файлов (**2**) выполняется еще несколько методов.

Метод **recoverFile** восстанавливает данные из файла, включая его название и расширение, изначально представленного в виде строки, содержащей base64-код байтов файла. Восстановленный файл отображен в объекте **InputFile**.

Метод **sendFile** занимается непосредственной отправкой файла в чат пользователю. На данном методе завершается выполнение операции 2.

Метод **notify** занимается информированием пользователя о результатах, последовавших после совершения каких-либо действий им или ботом. Частный случай такого метода ‒ это метод **listCommands**, который показывает список доступным пользователю операций.

Операции 0, 1, 2 и 3 завершаются оповещением пользователя об успешном ее выполнении или о возникших в ходе выполнения ошибках.

**/boot/CallbackResponseAction**

Представляет собой вспомогательный класс типа **Enumeration**, содержащий два объекта: GET и DELETE, соответствующие двум операциям (**2** и **3**). Он используется при реализации тех вышеуказанных методов, которые являются общими для двух указанных операций, для принятия решения о выполнении одной из них.

**/boot/Main**

Основной класс клиента, который запускает бота. После успешного запуска бот готов получать запросы и отправлять обновления (**Update**), которые потом обрабатываются в классе **DagaevProjectTelegramBot**. Реализован в виде одной функции **main**.

**/api/ZakharovAPI**

Интерфейс, описывающий, какие методы может использовать класс-обработчик (**DagaevProjectTelegramBot**). Назван в честь студента, который занимался разработкой серверной части приложения. Название методов полностью соответствует реализуемым операциям 0, 1, 2 и 3 за исключением одного вспомогательного, который занимается получением названий файлов. Подробно эти методы рассматриваются в классе, реализующем данный интерфейс.

**/api/SimpleZakharovAPI**

Класс реализует указанный выше интерфейс.

Здесь реализуются удаленные вызовы процедур. Каждый метод класса занимается отправлением и/или приемом потока данных посредством HTTP-соединения с сервером приложения, которые используются в обработчиках событий для выполнения запрашиваемых операций. В качества формата обмениваемых сообщений используется JSON. Каждому методу класса поставлен в соответствие свой **эндпоинт** (URI), по которому клиент обращается к серверу за получением очередного потока данных. В таблице 4 приведены эндпоинты для каждого метода. В силу того, что сеть, в рамках которой происходит взаимодействие клиента и сервера, динамически выделяет IP-адреса, в таблице не приводится конкретный IP-адрес сервера.

|  |  |  |
| --- | --- | --- |
| **№** | **Метод** | **Эндпоинт** |
| **0** | authenticate | http://<ip-адрес>:8080/v1/api/auth |
| **1** | Send | http://<ip-адрес>:8080/v1/api/send\_file |
| **2** | getFilesData | http://<ip-адрес>:8080/v1/api/fileById/<user\_id> |
| **3** | retrieve | http://<ip-адрес>:8080/v1/api/get\_file/<file\_id> |
| **4** | delete | http://<ip-адрес>:8080/v1/api/delete\_file/<file\_id> |

Таблица 4. Таблица соответствий между методами и эндпоинтами

Метод **authenticate** отправляет на сервер информацию по указанному в таблице 4 эндпоинту для учета аккаунтов пользователей, взаимодействующих в ботом. Для создания учетной записи необходим id пользователя в телеграме, а также имя пользователя.

|  |  |
| --- | --- |
| **HTTP-запрос (POST)** | **HTTP-ответ** |
| **Заголовок** | |
| Content-Type: application/json  Accept: application/json  Content-Length: <integer> | Status: 200/400  Content-Type: application/json  Accept: application/json |
| **Тело** | |
| [  { "telegram\_id": "..." },  { "username": "..." }  ] | - |

Таблица 5. Параметры HTTP для метода 0

Метод **send** отправляет JSON, содержащий необходимые данные для файла, который пользователь хочет сохранить в базе данных приложения. Структура JSON-объекта приведена в таблице 6.

|  |  |
| --- | --- |
| **HTTP-запрос (POST)** | **HTTP-ответ** |
| **Заголовок** | |
| Content-Type: application/json  Accept: application/json  Content-Length: <integer> | Status: 200/400  Content-Type: application/json  Accept: application/json |
| **Тело** | |
| [  { "size": "..." },  { "filename": "..." },  { "uploaded\_at": "..." },  { "message\_id": "..." },  { "user\_id": "..." },  { "description": "..." },  { "content": "..." } (**base64str**)  ] | - |

Таблица 6. Параметры HTTP для метода 1

Метод **getFilesData** отправляет HTTP-запрос без тела, однако передает id пользователя в качестве параметра URL. На выходе приходит массив, содержащий информацию о всех файлах, которые имеет указанный пользователь. Эта информация будет использована для восстановления нужных файлов перед его отправкой пользователю.

|  |  |
| --- | --- |
| **HTTP-запрос (GET)** | **HTTP-ответ** |
| **Заголовок** | |
| Content-Type: application/json  Accept: application/json | Status: 200/400  Content-Type: application/json  Accept: application/json  Content-Length: <integer> |
| **Параметры запроса** | |
| user\_id | |
| **Тело** | |
| - | [  [  {"id": "..."},  {"filename": "..."}  ],  [  {"id": "..."},  {"filename": "..."}  ],  ...  ] |

Таблица 7. Параметры HTTP для метода 2

Метод **retrieve** предназначен для получения файла по его id в базе данных. Здесь также не используется тело запроса, однако тело ответа содержит полную информацию о файле, который когда-то был сохранен в базе с помощью метода **send**.

|  |  |
| --- | --- |
| **HTTP-запрос (GET)** | **HTTP-ответ** |
| **Заголовок** | |
| Content-Type: application/json  Accept: application/json | Status: 200/400  Content-Type: application/json  Accept: application/json  Content-Length: <integer> |
| **Параметры запроса** | |
| file\_id | |
| **Тело** | |
| - | [  { "size": "..." },  { "filename": "..." },  { "uploaded\_at": "..." },  { "message\_id": "..." },  { "user\_id": "..." },  { "description": "..." },  { "content": "..." } (**base64str**)  ] |

Таблица 8. Параметры HTTP для метода 3

Метод **delete** обращается к серверу для удаления файла по его id базы данных.

|  |  |
| --- | --- |
| **HTTP-запрос (GET)** | **HTTP-ответ** |
| **Заголовок** | |
| Content-Type: application/json  Accept: application/json | Status: 200/400  Content-Type: application/json  Accept: application/json |
| **Параметры запроса** | |
| file\_id | |
| **Тело** | |

Таблица 9. Параметры HTTP для метода 4

# **2.4) Инструкция для запуска**

Ссылка на код клиента:

<https://github.com/ivansed2004/telegram_bot/tree/master>

Ссылка на код сервера:

<https://github.com/Sugyk/cloud_storage>

Для запуска серверной части нужно в корневой директории проекта ввести команду:

**$ go mod tidy**

Эта команда запускает автоматическую установку всех зависимостей. После этого следует запустить docker контейнер PostgreSQL, ввести в конфиг файл данные для подключения к базе данных и ввести команду

**$ make migrate**

Эта команда выполнит миграции и подготовит базу данных для работы сервера.

Далее команда запускающая сервер

**$ go run cmd/app/main.go**

Сервер запущен, чтобы отключить его ввести в консоль ctrl + C.

Для запуска клиентской части:.

1. Скачать ветку *master* проекта с гитхаба:

[https://github.com/ivansed2004/telegram\_bot](https://vk.com/away.php?to=https://github.com/ivansed2004/telegram_bot&utf=1)

2. Разархивировать проект;

3. Установить систему сборку Maven;

4. Совершить сборку проекта, используя систему Maven. В терминале перейти в корневую директорию проекта и ввести команду 'mvn package'. В корне проекта в папке target появяится два .jar-файла;

5. Создать отдельную папку, поместить туда .jar-файл **DagaevProject-1.0-SNAPSHOT.jar**;

6. Установить Docker;

7. Создать файл с названием Dockerfile со следующим содержанием:

**FROM openjdk:11-jre-slim**

**WORKDIR /run**

**COPY . /run/**

**CMD ["java", "-jar", "DagaevProject-1.0-SNAPSHOT.jar"]**

8. Запустить сборку образа с помощью команды:

**docker build -t tgbot:latest <путь к папке с Dockerfile>**

9. Запустить контейнер на основе созданного образа:

**docker run -p 8080:80 tgbot:latest**

Клиент запущен, чтобы отключить его ввести в консоль Ctrl + C.

**2.5) Инструкция пользователя**

1. Зайти в Телеграмм-бота и ввести команду **/start**. Сообщение, получаемое пользователем в ответ, содержит список доступных операций.

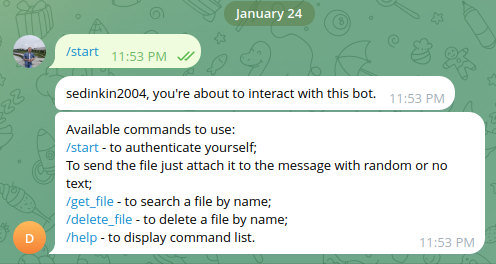


Рис. 5. Аутентификация пользователя

2. Для сохранение файла в облаке необходимо отправить в чат бота сохраняемый файл. При успешном сохранении файла бот отправляет соответствующее оповещение.

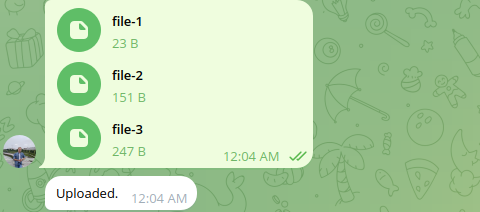


Рис. 6. Сохранение файлов в облаке

3. Для получения нужных файлов необходимо ввести команду **/get\_file**. В качестве ответа бот отображает клавиатуру, каждая кнопка которой содержит название файла.

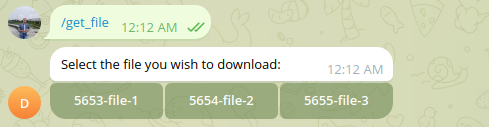


Рис. 7. Выбор файлов для скачивания

4. После нажатия на соответствующую кнопку бот отправит файл в чат.

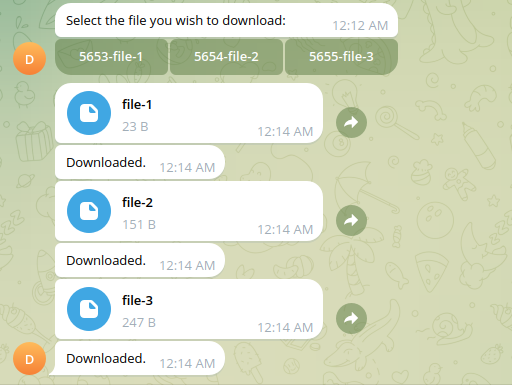


Рис. 8. Полученные пользователем файлы из облака

5. Для удаления файлов из облачного хранилища необходимо воспользоваться командой **/delete\_file**. Как и в случае с получением файлов, пользователь осуществляет выбор через предоставленную клавиатуру.

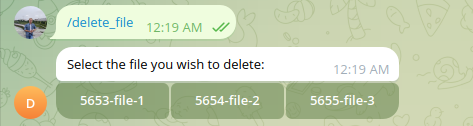
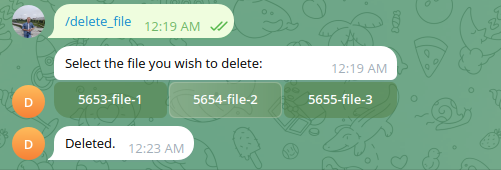


Рис. 9. Выбор файлов для удаления.

6. После нажатия кнопки пользователем бот производит удаление файла. На рис. 10 пользователь удаляет файл 5654-file-2.



# 

# Рис. 10. Удаление файла ботом.

При последующем выводе клавиатуры удалённый файл отсутствует, поскольку система не находит его в облачном хранилище.

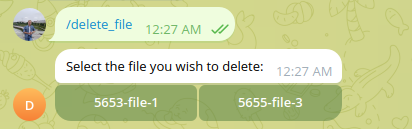


Рис. 11. Результат удаления файлов.

# **Выводы**

После успешной разработки клиент-серверного приложения для хранения данных в облаке с использованием REST на Go и клиента Telegram были достигнуты следующие результаты:

● **Было изучено создание клиент-серверных приложений на Go**: В ходе разработки проекта было изучено, как строить клиент-серверные приложения с использованием языка Go, а также как интегрировать их с внешними сервисами через REST API. Этот опыт был полезен для разработки эффективных и масштабируемых решений.

● **Реализация основной функциональности**: Разработано приложение, которое позволяет пользователям загружать, удалять и получать файлы с облачного хранилища через REST API. Также был реализован клиент для Telegram, обеспечивающий взаимодействие с сервером через Telegram-бота, позволяя пользователям отправлять команды и получать статус файлов.

● **Использование технологий**: Для серверной части был использован язык Go, а также библиотеки для работы с REST API и базой данных. Для взаимодействия с Telegram использовался официальный Telegram API, а для хранения данных — облачная система с поддержкой REST.

● **Работоспособность**: Проведено обширное тестирование функциональности как серверной части приложения, так и Telegram-бота. Все функции — загрузка файлов, удаление, просмотр списка файлов и взаимодействие через Telegram — успешно работают. Приложение прошло тесты на различных платформах и операционных системах, обеспечивая стабильную работу в различных окружениях.

● **Интеграция и документация**: Предоставлено подробное руководство по использованию приложения и Telegram-бота. Включены инструкции по настройке серверной части, а также описание работы с ботом в Telegram. Документация также объясняет основные функции API и взаимодействие с ним через различные команды бота.

● **Перспективы развития**: В планах дальнейшего развития проекта - добавление новых функций, таких как поддержка различных типов файловых хранилищ и улучшение обработки ошибок. Также планируется улучшение интерфейса Telegram-бота, добавление новых команд для управления файлами и расширение возможностей API для более гибкой работы с данными. В будущем будет рассматриваться интеграция с другими облачными сервисами для обеспечения масштабируемости.

В целом, разработанное клиент-серверное приложение для хранения данных в облаке с использованием REST и Telegram-бота представляет собой удобный инструмент для загрузки, удаления и получения файлов через два интерфейса: REST API и Telegram. Определены дальнейшие планы для улучшения и расширения проекта.

# **Список используемых источников**

1. Donovan, A. A., & Kernighan, B. W. (2015). *The Go Programming Language*. Addison-Wesley.
2. Chang, S. S. (2016). *Go Web Programming*. Packt Publishing.
3. Telegram Bot API Documentation. (2021). *Telegram API Documentation*. <https://core.telegram.org/bots/api>
4. Richardson, L., Amundsen, M., & Ruby, S. (2013). *RESTful Web APIs*. O'Reilly Media.
5. Bodner, J. (2017). *Learning Go*. O'Reilly Media.
6. Hernandez, M. J. (2013). *Database Design for Mere Mortals*. Addison-Wesley.
7. Forta, B. (2018). *SQL in 10 Minutes, Sams Teach Yourself*. Sams Publishing.
8. Richardson, C. (2018). *Microservices Patterns: With Examples in Java*. Manning Publications.
9. Arundel, J., & Domingus, J. (2019). *Cloud Native DevOps with Kubernetes*. O'Reilly Media.
10. Pike, R. (2020). *Practical Go Lessons*. O'Reilly Media.
11. Yellavula, N. (2020). *Building Scalable Web Applications Using Cloud-Native Architecture*. Packt Publishing.
12. Newman, S. (2015). *Building Microservices*. O'Reilly Media.

# **Приложение**

**Листинг Java-кода клиента:**

**/boot/CallbackResponseAction**

*package ru.ikvt.dagprojbot.boot;*

*public enum CallbackResponseAction {*

*GET,*

*DELETE*

*}*

**/boot/DagaevProjectTelegramBot**

*package ru.ikvt.dagprojbot.boot;*

*import org.json.JSONArray;*

*import org.json.JSONObject;*

*import org.slf4j.Logger;*

*import org.slf4j.LoggerFactory;*

*import org.telegram.telegrambots.bots.TelegramLongPollingBot;*

*import org.telegram.telegrambots.meta.api.methods.GetFile;*

*import org.telegram.telegrambots.meta.api.methods.send.SendDocument;*

*import org.telegram.telegrambots.meta.api.methods.send.SendMessage;*

*import org.telegram.telegrambots.meta.api.objects.Document;*

*import org.telegram.telegrambots.meta.api.objects.File;*

*import org.telegram.telegrambots.meta.api.objects.InputFile;*

*import org.telegram.telegrambots.meta.api.objects.Update;*

*import org.telegram.telegrambots.meta.api.objects.replykeyboard.InlineKeyboardMarkup;*

*import org.telegram.telegrambots.meta.api.objects.replykeyboard.buttons.InlineKeyboardButton;*

*import org.telegram.telegrambots.meta.exceptions.TelegramApiException;*

*import ru.ikvt.dagprojbot.api.SimpleZakharovAPI;*

*import ru.ikvt.dagprojbot.api.ZakharovAPI;*

*import java.io.\*;*

*import java.net.HttpURLConnection;*

*import java.net.URL;*

*import java.util.ArrayList;*

*import java.util.Base64;*

*import java.util.List;*

*import java.util.Objects;*

*public class DagaevProjectTelegramBot extends TelegramLongPollingBot {*

*private static final Logger LOGGER = LoggerFactory.getLogger(DagaevProjectTelegramBot.class);*

*private final String NAME;*

*private final String TOKEN;*

*private final ZakharovAPI API;*

*public DagaevProjectTelegramBot( String name, String token ) {*

*super(token);*

*this.NAME = name;*

*this.TOKEN = token;*

*this.API = new SimpleZakharovAPI();*

*}*

*@Override*

*public void onUpdateReceived( Update update ) {*

*LOGGER.info("Event received.");*

*if ( update.hasMessage() ) {*

*try {*

*onMessageReceived( update );*

*} catch ( Exception ex ) {*

*LOGGER.error("onMessageReceived. Something went wrong when connecting the server.");*

*}*

*} else if ( update.hasCallbackQuery() ) {*

*try {*

*onButtonClicked( update );*

*} catch ( Exception ex ) {*

*LOGGER.error("onButtonClicked. Something went wrong when connecting the server.");*

*}*

*}*

*}*

*private void onMessageReceived( Update update ) throws Exception {*

*long chatId = update.getMessage().getChatId();*

*LOGGER.info("Message detected in event received.");*

*if ( update.getMessage().hasText() && !update.getMessage().hasDocument() ) {*

*String action = update.getMessage().getText();*

*switch ( action ) {*

*case "/start":*

*LOGGER.info("/start operation is selected.");*

*auth( update );*

*listCommands( chatId );*

*break;*

*case "/get\_file":*

*LOGGER.info("/get\_file operation is selected.");*

*preprocessKeyboard( update, CallbackResponseAction.GET );*

*break;*

*case "/delete\_file":*

*LOGGER.info("/delete\_file operation is selected.");*

*preprocessKeyboard( update, CallbackResponseAction.DELETE );*

*break;*

*case "/help":*

*LOGGER.info("/help operation is selected.");*

*listCommands( chatId );*

*break;*

*default:*

*LOGGER.info("Unexpected command selected.");*

*notify( chatId, "Incorrect command. Choose one from the following list: " );*

*listCommands( chatId );*

*}*

*} else if ( update.getMessage().hasDocument() ) {*

*LOGGER.info("/send\_file operation is selected.");*

*send( update );*

*notify( chatId, "Uploaded." );*

*}*

*}*

*private void onButtonClicked( Update update ) throws Exception {*

*if ( update.hasCallbackQuery() ) {*

*String[] callbackArray = update.getCallbackQuery().getData().split("-");*

*long fileId = Long.parseLong( callbackArray[0] );*

*long chatId = Long.parseLong( callbackArray[2] );*

*CallbackResponseAction action =*

*( Objects.equals(callbackArray[1], "download") ) ?*

*CallbackResponseAction.GET : CallbackResponseAction.DELETE;*

*respondCallback( chatId, fileId, action );*

*if ( action == CallbackResponseAction.GET ) {*

*notify( chatId, "Downloaded." );*

*} else {*

*notify( chatId, "Deleted." );*

*}*

*}*

*}*

*private void auth( Update update ) throws Exception {*

*long chatId = update.getMessage().getChatId();*

*String username = update.getMessage().getFrom().getUserName();*

*API.authenticate( chatId, username );*

*notify( chatId, username + ", you're about to interact with this bot. " );*

*}*

*private void send( Update update ) throws Exception {*

*long messageId = update.getMessage().getMessageId();*

*long userId = update.getMessage().getFrom().getId();*

*Document document = update.getMessage().getDocument();*

*String fileName = document.getFileName();*

*String base64str = getBase64str( document );*

*long size = document.getFileSize();*

*API.send( userId, messageId, fileName, size, base64str );*

*}*

*private void preprocessKeyboard( Update update, CallbackResponseAction action ) throws Exception {*

*long userId = update.getMessage().getFrom().getId();*

*long chatId = update.getMessage().getChatId();*

*JSONObject files = API.getFilesData( userId );*

*setButtons( chatId, files, action );*

*}*

*private void sendFile( long chatId, InputFile fileToSend ) {*

*SendDocument sendDocument = new SendDocument();*

*sendDocument.setChatId( String.valueOf(chatId) );*

*sendDocument.setDocument( fileToSend );*

*try {*

*execute(sendDocument);*

*} catch (Exception e) {*

*LOGGER.error("Unable to send file to chat.");*

*}*

*}*

*private InputFile recoverFile( String filename, String base64str ) {*

*ByteArrayInputStream stream = null;*

*try {*

*byte[] data = Base64.getDecoder().decode(base64str);*

*stream = new ByteArrayInputStream( data );*

*} catch (IllegalArgumentException e) {*

*LOGGER.error("Error occurred while decoding the base64 string.");*

*}*

*return new InputFile( stream, filename );*

*}*

*private void respondCallback( long chatId, long fileId, CallbackResponseAction action ) throws Exception {*

*if ( action == CallbackResponseAction.GET ) {*

*JSONObject json = API.retrieve( fileId );*

*String filename = (String) json.get("filename");*

*String base64str = (String) json.get("file\_body");*

*InputFile fileToSend = recoverFile( filename, base64str );*

*sendFile( chatId, fileToSend );*

*} else {*

*API.delete( fileId );*

*}*

*}*

*private void setButtons( long chatId, JSONObject files, CallbackResponseAction action ) {*

*SendMessage sendMessage = new SendMessage();*

*sendMessage.setChatId( chatId );*

*String text = (action == CallbackResponseAction.GET) ? "download" : "delete";*

*sendMessage.setText("Select the file you wish to " + text + ":");*

*InlineKeyboardMarkup markup = new InlineKeyboardMarkup();*

*List<List<InlineKeyboardButton>> keyboard = new ArrayList<>();*

*List<InlineKeyboardButton> buttons = new ArrayList<>();*

*JSONArray filesArray = files.getJSONArray("files");*

*for ( Object obj : filesArray ) {*

*InlineKeyboardButton button = new InlineKeyboardButton();*

*JSONObject json = (JSONObject) obj;*

*String buttonText = (String) json.get("filename");*

*String callback = String.valueOf(json.get("id"));*

*button.setText( callback + "-" + buttonText );*

*button.setCallbackData( callback + "-" + text + "-" + chatId );*

*buttons.add(button);*

*}*

*keyboard.add(buttons);*

*markup.setKeyboard(keyboard);*

*sendMessage.setReplyMarkup( markup );*

*try {*

*execute(sendMessage);*

*} catch (Exception e) {*

*LOGGER.error("Unable to display keyboard to chat.");*

*}*

*}*

*private String getBase64str(Document document) {*

*String base64str = null;*

*HttpURLConnection connection = null;*

*String fileId = document.getFileId();*

*try {*

*GetFile getFile = new GetFile();*

*getFile.setFileId(fileId);*

*File file = execute(getFile);*

*String filePath = file.getFilePath();*

*String endpoint = "https://api.telegram.org/file/bot" + TOKEN + "/" + filePath;*

*try {*

*URL url = new URL(endpoint);*

*connection = (HttpURLConnection) url.openConnection();*

*connection.setRequestMethod("GET");*

*try (InputStream inputStream = connection.getInputStream()) {*

*ByteArrayOutputStream byteArrayOutputStream = new ByteArrayOutputStream();*

*byte[] buffer = new byte[4096]; // 1024*

*int bytesRead;*

*while ( (bytesRead = inputStream.read(buffer)) != -1 ) {*

*byteArrayOutputStream.write(buffer, 0, bytesRead);*

*}*

*base64str = Base64*

*.getEncoder()*

*.encodeToString( byteArrayOutputStream.toByteArray() );*

*byteArrayOutputStream.close();*

*}*

*if (connection.getResponseCode() == HttpURLConnection.HTTP\_OK) {*

*LOGGER.info("Successful bytes-base64str transforming.");*

*} else {*

*LOGGER.error("Non OK HTTP Status. Something went wrong while connecting the server.");*

*}*

*} catch ( Exception ex ) {*

*LOGGER.error("Something went wrong while connecting the server.");*

*} finally {*

*connection.disconnect();*

*LOGGER.info("GetBase64String. Connection interrupted.");*

*}*

*} catch ( Exception ex ) {*

*LOGGER.error("Something went wrong while preparing the file.");*

*}*

*return base64str;*

*}*

*private void listCommands( long chatId ) {*

*notify( chatId, "Available commands to use: \n" +*

*"/start - to authenticate yourself; \n" +*

*"To send the file just attach it to the message with random or no text; \n" +*

*"/get\_file - to search a file by name; \n" +*

*"/delete\_file - to delete a file by name; \n" +*

*"/help - to display command list." );*

*}*

*private void notify( long chatId, String text ) {*

*SendMessage message = new SendMessage();*

*message.setChatId(chatId);*

*message.setText(text);*

*try {*

*execute(message);*

*} catch (TelegramApiException e) {*

*LOGGER.error("Unable to send message to chat.");*

*}*

*}*

*@Override*

*public String getBotUsername() {*

*return NAME;*

*}*

*}*

**/boot/Main**

*package ru.ikvt.dagprojbot.boot;*

*import org.slf4j.Logger;*

*import org.slf4j.LoggerFactory;*

*import org.telegram.telegrambots.meta.TelegramBotsApi;*

*import org.telegram.telegrambots.meta.exceptions.TelegramApiException;*

*import org.telegram.telegrambots.updatesreceivers.DefaultBotSession;*

*public class Main {*

*private static final Logger LOGGER = LoggerFactory.getLogger(Main.class);*

*public static void main(String[] args) {*

*String name = "dagprojbot";*

*String token = "7933878576:AAGd8YCQIfop1mI0lY3x1eeji\_zKN55M1pQ";*

*DagaevProjectTelegramBot bot = new DagaevProjectTelegramBot(name, token);*

*try {*

*TelegramBotsApi api = new TelegramBotsApi(DefaultBotSession.class);*

*api.registerBot(bot);*

*LOGGER.info("Bot is running.");*

*} catch (TelegramApiException e) {*

*LOGGER.error("Something went wrong while trying the bot to boot up");*

*System.out.println( e.getMessage() );*

*}*

*}*

*}*

**/api/SimpleZakharovAPI**

*package ru.ikvt.dagprojbot.api;*

*import org.json.JSONObject;*

*import org.slf4j.Logger;*

*import org.slf4j.LoggerFactory;*

*import java.io.\*;*

*import java.net.HttpURLConnection;*

*import java.net.URL;*

*import java.nio.charset.StandardCharsets;*

*public class SimpleZakharovAPI implements ZakharovAPI {*

*private static final Logger LOGGER = LoggerFactory.getLogger(SimpleZakharovAPI.class);*

*String prefix = "http://192.168.43.149:8080/v1/api";*

*@Override*

*public void authenticate( long userId, String username ) throws Exception {*

*JSONObject authRequest = new JSONObject();*

*authRequest.put("telegram\_id", userId);*

*authRequest.put("username", username);*

*String jsonString = authRequest.toString();*

*byte[] data = jsonString.getBytes( StandardCharsets.UTF\_8 );*

*HttpURLConnection connection = null;*

*String endpoint = prefix + "/auth";*

*try {*

*URL url = new URL(endpoint);*

*connection = (HttpURLConnection) url.openConnection();*

*connection.setDoOutput(true);*

*connection.setRequestMethod("POST");*

*connection.setRequestProperty("Content-Type", "application/json");*

*connection.setRequestProperty("Accept", "application/json");*

*connection.setRequestProperty("Content-Length", String.valueOf( data.length ));*

*try (OutputStream outputStream = connection.getOutputStream()) {*

*outputStream.write( data, 0, data.length );*

*} catch (IOException ex) {*

*throw new Exception("Non OK HTTP Status. Something went wrong while connecting the server.");*

*}*

*if ( connection.getResponseCode() == HttpURLConnection.HTTP\_OK ) {*

*LOGGER.info("Authenticate. Authentication successful");*

*}*

*} catch (IOException ex) {*

*LOGGER.error("Non OK HTTP Status. Something went wrong while connecting the server.");*

*} finally {*

*connection.disconnect();*

*LOGGER.info("Authenticate. Connection interrupted.");*

*}*

*}*

*@Override*

*public void send( long userId, long messageId, String fileName, long size, String base64str ) throws Exception {*

*JSONObject authRequest = new JSONObject();*

*authRequest.put("file\_size", size);*

*authRequest.put("filename", fileName);*

*// authRequest.put("uploaded\_at", LocalDateTime.now().format(DateTimeFormatter.ofPattern("yyyy:MM:dd mm:HH")));*

*authRequest.put("message\_id", messageId);*

*authRequest.put("user\_id", userId);*

*authRequest.put("description", "<empty>");*

*authRequest.put("file\_body", base64str);*

*String jsonString = authRequest.toString();*

*byte[] data = jsonString.getBytes( StandardCharsets.UTF\_8 );*

*HttpURLConnection connection = null;*

*String endpoint = prefix + "/send\_file";*

*try {*

*URL url = new URL(endpoint);*

*connection = (HttpURLConnection) url.openConnection();*

*connection.setDoOutput(true);*

*connection.setRequestMethod("POST");*

*connection.setRequestProperty("Content-Type", "application/json");*

*connection.setRequestProperty("Accept", "application/json");*

*connection.setRequestProperty("Content-Length", String.valueOf( data.length ));*

*try (OutputStream outputStream = connection.getOutputStream()) {*

*outputStream.write( data, 0, data.length );*

*} catch (IOException ex) {*

*throw new Exception("Non OK HTTP Status. Something went wrong while connecting the server.");*

*}*

*if ( connection.getResponseCode() == HttpURLConnection.HTTP\_OK ) {*

*LOGGER.info("Send. File sent.");*

*}*

*} catch (IOException ex) {*

*ex.printStackTrace();*

*} finally {*

*connection.disconnect();*

*LOGGER.info("Send. Connection interrupted.");*

*}*

*}*

*@Override*

*public JSONObject getFilesData( long userId ) throws Exception {*

*StringBuilder sb = new StringBuilder();*

*HttpURLConnection connection = null;*

*String endpoint = prefix + "/get\_files/" + userId;*

*try {*

*URL url = new URL(endpoint);*

*connection = (HttpURLConnection) url.openConnection();*

*connection.setDoInput(true);*

*connection.setRequestMethod("GET");*

*connection.setRequestProperty("Content-Type", "application/json");*

*connection.setRequestProperty("Accept", "application/json");*

*try (InputStream inputStream = connection.getInputStream()) {*

*BufferedReader reader = new BufferedReader(new InputStreamReader(inputStream));*

*String line;*

*while ( (line = reader.readLine()) != null ) {*

*sb.append(line);*

*}*

*}*

*if ( connection.getResponseCode() == HttpURLConnection.HTTP\_OK ) {*

*LOGGER.info("GetFilesData. Files data received.");*

*} else {*

*throw new Exception("Non OK HTTP Status. Something went wrong while connecting the server.");*

*}*

*} catch (IOException ex) {*

*ex.printStackTrace();*

*} finally {*

*connection.disconnect();*

*LOGGER.info("GetFilesData. Connection interrupted.");*

*}*

*return new JSONObject( sb.toString() );*

*}*

*@Override*

*public JSONObject retrieve( long fileId ) throws Exception {*

*StringBuilder sb = new StringBuilder();*

*HttpURLConnection connection = null;*

*String endpoint = prefix + "/get\_file/" + fileId;*

*try {*

*URL url = new URL(endpoint);*

*connection = (HttpURLConnection) url.openConnection();*

*connection.setDoInput(true);*

*connection.setRequestMethod("GET");*

*connection.setRequestProperty("Content-Type", "application/json");*

*connection.setRequestProperty("Accept", "application/json");*

*try (InputStream inputStream = connection.getInputStream()) {*

*BufferedReader reader = new BufferedReader(new InputStreamReader(inputStream));*

*String line;*

*while ( (line = reader.readLine()) != null ) {*

*sb.append(line);*

*}*

*}*

*if ( connection.getResponseCode() == HttpURLConnection.HTTP\_OK ) {*

*LOGGER.info("Get. File received.");*

*} else {*

*throw new Exception("Non OK HTTP Status. Something went wrong while connecting the server.");*

*}*

*} catch (IOException ex) {*

*ex.printStackTrace();*

*} finally {*

*connection.disconnect();*

*LOGGER.info("Get. Connection interrupted.");*

*}*

*return new JSONObject( sb.toString() );*

*}*

*@Override*

*public void delete( long fileId ) throws Exception {*

*HttpURLConnection connection = null;*

*String endpoint = prefix + "/delete\_file/" + fileId;*

*try {*

*URL url = new URL(endpoint);*

*connection = (HttpURLConnection) url.openConnection();*

*connection.setRequestMethod("GET");*

*connection.setRequestProperty("Content-Type", "application/json");*

*connection.setRequestProperty("Accept", "application/json");*

*if ( connection.getResponseCode() != HttpURLConnection.HTTP\_OK ) {*

*throw new Exception("Non OK HTTP Status. Something went wrong while connecting the server.");*

*}*

*LOGGER.info("Delete. File deleted.");*

*} catch (IOException ex) {*

*ex.printStackTrace();*

*} finally {*

*connection.disconnect();*

*LOGGER.info("Delete. Connection interrupted.");*

*}*

*}*

*}*

**/api/ZakharovAPI**

*package ru.ikvt.dagprojbot.api;*

*import org.json.JSONObject;*

*public interface ZakharovAPI {*

*void authenticate( long userId, String username ) throws Exception;*

*void send( long userId, long messageId, String fileName, long size, String base64str ) throws Exception;*

*JSONObject getFilesData( long userId ) throws Exception;*

*JSONObject retrieve( long fileId ) throws Exception;*

*void delete( long fileId ) throws Exception;*

*}*