УДК 004.9

РАЗРАБОТКА СЕРВЕРНОЙ ЧАСТИ ВЕБ-СЕРВИСА СИСТЕМЫ ОБЛАЧНОГО ХРАНЕНИЯ ФАЙЛОВ

Брагин Иван Игоревич[1], Мельникова Ольга Игоревна[2]

[1]Студент;
Государственный университет «Дубна»;
Россия, 141980, Московская обл., г. Дубна, ул. Университетская, 19;
e-mail: bragin_ivan02@mail.ru.

[2]Кандидат технических наук, доцент;
Государственный университет «Дубна»;
Россия, 141980, Московская обл., г. Дубна, ул. Университетская, 19;
e-mail: oimelnik@mail.ru.

Данная работа посвящена разработке и реализации веб-приложения облачного хранилища файлов. Целью являлось создание удобного и функционального инструмента для хранения и управления файлами в облаке. В работе проводится анализ существующих облачных хранилищ, выявляются их основные особенности и недостатки. На основе этого анализа определяются требования к разрабатываемому приложению, проектируется архитектура приложения, выбор технологий и инструментов разработки. В ходе реализации уделяется особое внимание аспектам масштабируемости и отказоустойчивости системы. В результате создается полнофункциональное веб-приложение, позволяющее пользователям загружать, хранить и организовывать файлы в облаке.

Ключевые слова: облачное хранилище, веб-приложение, файлы.

Для цитирования:
Брагин И. И., Мельникова О. И. Разработка серверной части веб-сервиса системы облачного хранения файлов // Системный анализ в науке и образовании: сетевое научное издание. 2024. № 4. С. 25-36. EDN: GGZYCQ. URL: https://sanse.ru/index.php/sanse/article/view/633.

DEVELOPMENT OF THE SERVER PART OF THE WEB SERVICE OF THE CLOUD FILE STORAGE SYSTEM

Bragin Ivan I.[1], Melnikova Olga I.[1]

[1]Student;
Dubna State University;
19 Universitetskaya Str., Dubna, Moscow region, 141980, Russia;
e-mail: bragin_ivan02@mail.ru.

[2]PhD in Engineering sciences, associate professor;
Dubna State University;
19 Universitetskaya Str., Dubna, Moscow region, 141980, Russia;
e-mail: oimelnik@mail.ru.

This paper presents are dedicated to the development and implementation of a web-based cloud file storage application. The goal of the work is to create a convenient and functional tool for storing and managing files in the cloud. The thesis includes an analysis of existing cloud storage solutions, identifying their key features and shortcomings. Based on this analysis, requirements for the developed application are determined. Then, the architecture of the application is designed, and technologies and development tools are chosen. Special attention is paid to aspects of scalability and system resilience during implementation. As a result of the work, a fully functional web application is created, allowing users to upload, store, and organize files in the cloud.

Keywords: cloud storage, web application, files.

For citation:
Bragin I. I., Melnikova O. I. Development of the server part of the web service of the cloud file storage system. *System analysis in science and education*, 2024;(4):25-36 (in Russ). EDN: GGZYCQ. Available from: https://sanse.ru/index.php/sanse/article/view/633.

Введение

В современном информационном обществе, характеризующемся стремительным развитием технологий, вопрос эффективного и безопасного хранения данных приобретает все большее значение. В этом контексте облачные технологии представляют собой неотъемлемую часть цифровой инфраструктуры, обеспечивая удобный доступ к информации и ее хранению на удаленных серверах. Существующие системы хранения файлов не всегда и не во всем устраивают пользователей. В данной работе будет рассмотрена концепция, проектирование и реализация ~~такого~~ веб-сервиса, который будет учитывать современные требования к безопасности, производительности и функциональности систем облачного хранения файлов.

1. Анализ существующих решений

В настоящее время существует множество решений для облачного хранения данных [1] с открытым исходным кодом и гибкой настройкой. Одними из самых популярных решений [2] являются *Owncloud, Seafile* [3], *Nextcloud*. Рассмотрим их более детально.

SeaFile:

- Преимущества:

1) Открытый исходный код: обеспечивает прозрачность и возможность адаптации под индивидуальные требования.

2) Энд-ту-энд шифрование: обеспечивает высокий уровень конфиденциальности данных.

3) Совместная работа: Поддержка командной работы, комментарии, история версий.

4) Документация: Наличие подробной документации для взаимодействия с системой.

- Недостатки:

1) Требует технической экспертизы: настройка и управление требуют определенного уровня технического опыта.

Owncloud:

- Преимущества:

1) Бесплатное использование: Бесплатная базовая версия, а также приложения для *Android* и *iOS*.

2) Доступ из любого места: Доступ к хранилищу из любого места, при наличии подключения к сети Интернет.

3) Синхронизация данных: Синхронизация файлов в клиенте для *Windows* с учетной записью *Owncloud*.

- Недостатки:

1) Производительность: Долгая работа веб-интерфейса при загрузке файлов.

2) Локализация: Отсутствие русского языка.

3) Ограниченный бесплатный доступ: Некоторые продвинутые функции доступны только на платных тарифах.

4) Требует технической экспертизы: Настройка и управление требуют определенного уровня технического опыта.

5) Документация: Отсутствие подробной документации для взаимодействия с системой.

Nextcloud:

- Преимущества:

1) Бесплатное пользование: Бесплатная базовая версия продукта.

2) Производительность: Быстрая и стабильная работа системы.

3) Документация: Наличие подробной документации для взаимодействия с системой.

- Недостатки:

1) Требует технической экспертизы: Настройка и управление требуют определенного уровня технического опыта.

2) Ограниченный бесплатный доступ: Некоторые продвинутые функции доступны только на платных тарифах. В настоящие время существуют иностранные готовые программные решения, которые можно было использовать при реализации системы.

Для наглядности результаты анализа были занесены в таблицу 1, где баллы выставлялись по следующей шкале оценок: «1» – соответствует критерию, «0.5» – частично соответствует критерию и «0» – не соответствует критерию.

Табл. 1. Сравнительный анализ систем

	SeaFile	***Owncloud***	***Nextcloud***
Бесплатное использование	1	0.5	0.5
Доступ из любого места	1	1	1
Производительность	1	0.5	1
Простота установки и настройки	0	0	0
Документация	1	0	1
Совместная работа	1	1	1

Безусловно, вышеприведенные неплохо справляются с задачей хранения файлов. Однако, они являются платными и не обладают простотой установки, что может быть сложно для пользователей на начальном этапе вхождения *IT*. Именно поэтому и интересно разработать свою систему облачного хранения небольших файлов для личного пользования.

На основе анализа этих решений видно, что успешные сервисы обеспечивают баланс между безопасностью, удобством использования и функциональностью. Разработка нового веб-сервиса системы облачного хранения файлов должна учитывать эти факторы, чтобы соответствовать ожиданиям разнообразных пользователей, а также иметь свои отличительные особенности:

- Полный контроль над данными: Собственное хранилище обеспечивает максимальную конфиденциальность, так как данные не передаются сторонним компаниям.
- Гибкость и кастомизация: Адаптация функционала под специфические потребности, а также возможность легкого расширения системы новыми функциями.
- Независимость: Собственное решение устраняет зависимость от иностранных компаний, их бизнес-политик и возможных изменений в условиях использования.

2. Требования к системе

По результатам проведенного анализа выделим необходимые функциональные требования:

1 Работа с пользователями:

1.1 Регистрация и аутентификация:

- Возможность регистрации новых пользователей с использованием электронной почты.
- Система аутентификации для входа в систему с использованием учетных данных пользователя.

1.2 Поддержка многопользовательского режима:

- Возможность добавления нескольких пользователей с уникальными учетными данными.

2 Работа с данными:

2.1 Управление файлами:

- Возможность загрузки файлов на сервер.
- Создание новых папок и организация файлов в структурированный вид.
- Возможность просмотра, редактирования и удаления файлов и папок.
- Поддержка различных типов файлов, включая документы, изображения, аудио и видео.

2.2 Поиск и фильтрация:

- Поиск файлов на основе ключевых слов, типов файлов, даты и других параметров.
- Фильтрация файлов по категориям, тегам или другим пользовательским параметрам.

3 Безопасность доступа к данным:

3.1 Контроль доступа:

- Установка прав доступа для файлов и папок (чтение, запись, удаление) на уровне пользователей и групп.
- Возможность предоставления временных и одноразовых разрешений на доступ к файлам.
- Аутентификация *API*-запросов для обеспечения безопасности.

4 Интерфейс взаимодействия и доступность:

4.1 Доступ к системе через *API*:

- Предоставление *API* для взаимодействия с сервисом, включая возможность загрузки, скачивания и управления файлами.
- Аутентификация *API*-запросов для обеспечения безопасности.

4.2 Облачная синхронизация и доступность:

- Автоматическая синхронизация файлов между различными устройствами пользователя.
- Возможность доступа к файлам из любой точки мира через интернет.

3. Архитектура системы

Диаграмма компонентов иллюстрирует взаимодействие пользователя с системой через *REST* [4] интерфейс. В свою очередь веб-сервис отправляет *SQL*-запросы в базу данных, а также сохраняет либо получает файлы из объектного хранилища. (см. рис. 1).

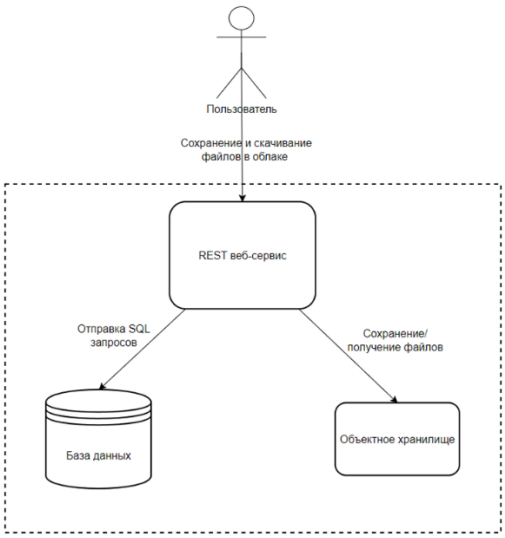

Рис. 1. Представление компонентов системы

Пользователь взаимодействует с *REST* интерфейсом приложения, отправляя запросы на сервер. Бэкэнд получив запросы, обращается к базе данных для получения необходимой метаинформации или к облачному хранилищу для сохранения или скачивания файлов. После обработки запросов бэкэнд отправляет ответы на клиент пользователя (см. рис. 2).

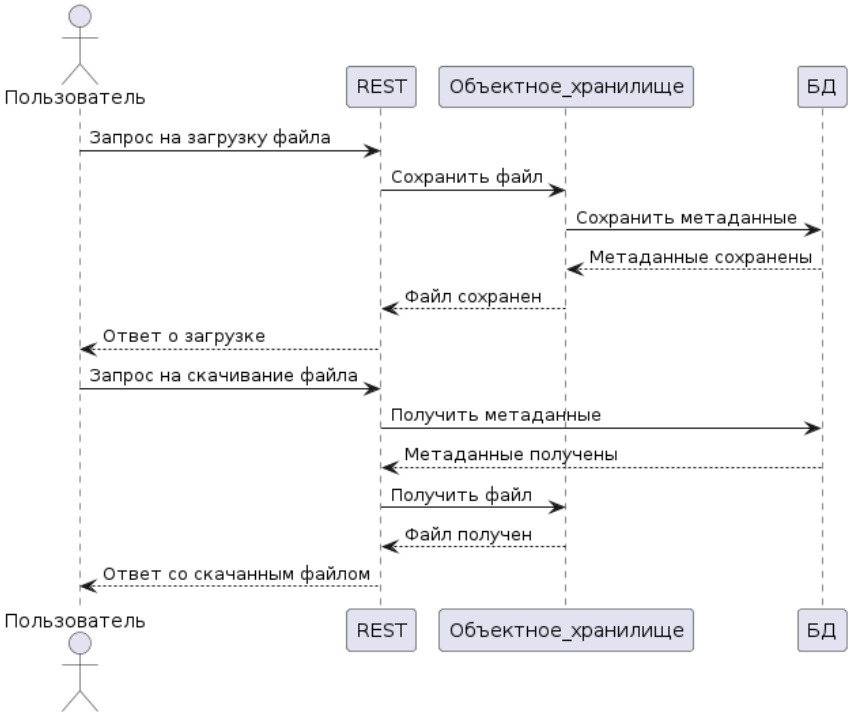

Рис. 2. Диаграмма последовательности. Загрузка файла.

На стадии проектирования были подготовлены описания баз данных и их логическая и физическая схемы. Ниже приведены основные таблицы баз данных (см. рис. 3-4)

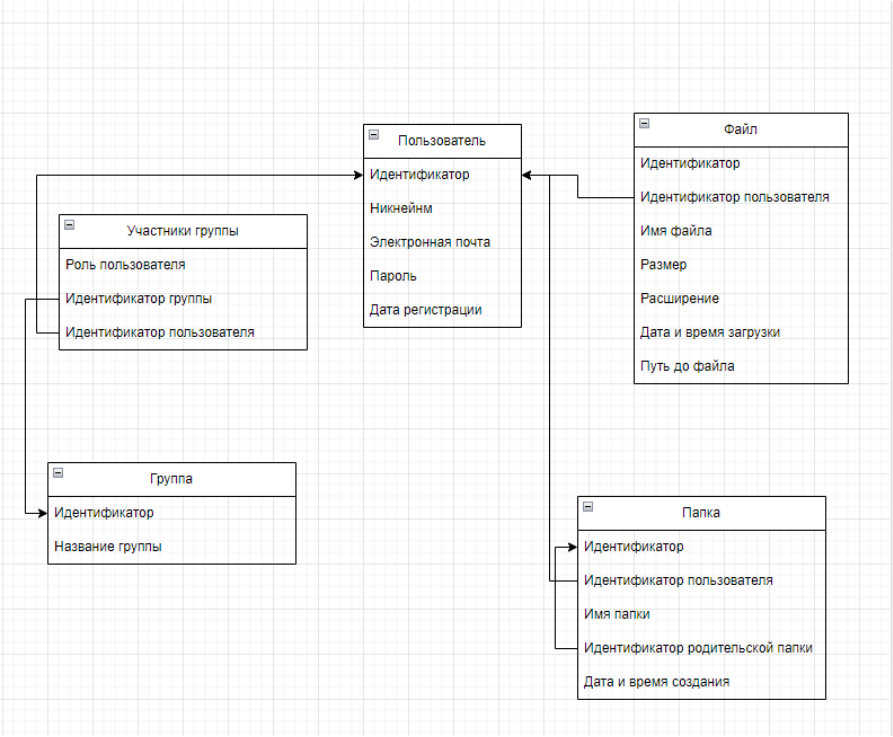

Рис. 3. Логическая схема базы данных

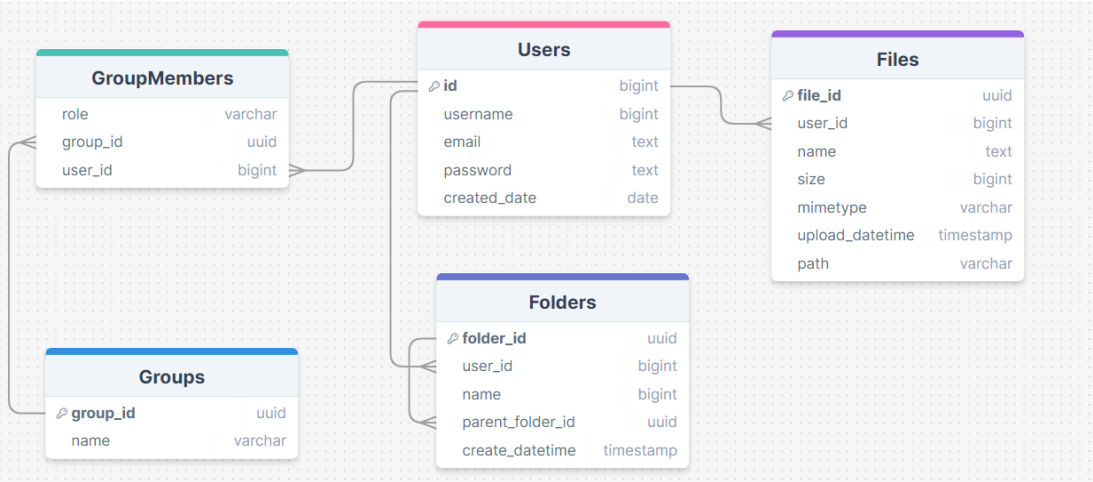

Рис. 4. Физическая схема базы данных

4. Реализация

Главная функция серверной части – принимать клиентские запросы и соответствующим образом обрабатывать, в зависимости от того, какой метод *API* был вызван пользователем.

Основной группой методов *API* (см. рис. 5) являются методы сохранения и получения файлов.

Эндпоинт — конечная точка веб-сервиса, к которой клиентское приложение обращается для выполнения определённых операций или получения данных

```
    }
}

func (fh *Handler) RegisterRoutes() func(r chi.Router) {  1 usage  i.bragin
    return func(r chi.Router) {
        r.Route( pattern: "/file", func(r chi.Router) {
            r.Post( pattern: "/add", fh.SaveFile())
            r.Get( pattern: "/{id}", fh.GetFile())
        })
    }
}
```

Рис. 5. Эндоинты для работы с файлами

Эндпоинт *POST /file/add* обрабатывает запросы на добавление файла в файловое хранилище. В методе SaveFile (см. рис. 6) валидируется пользовательский запрос на наличие файла и его наименования.

```go
func (fh *Handler) SaveFile() http.HandlerFunc { 1 usage  ± i.bragin *
    return func(w http.ResponseWriter, r *http.Request) {
        const op = "SaveFile Handler"

        log := fh.log.With(
            slog.String( key: "OP", op),
            slog.String( key: "request_id", middleware.GetReqID(r.Context())),
        )
        r.ParseMultipartForm( maxMemory: 0)
        files, ok := r.MultipartForm.File["file"]
        if !ok || len(files) == 0 {
            render.JSON(w, r, dto.CreateError( msg: "file can`t be empty"))
        }
        fileInfo := files[0]
        fileReader, _ := fileInfo.Open()

        createFileDto := dto.CreateFileDto{
            Name:   fileInfo.Filename,
            Size:   fileInfo.Size,
            Reader: fileReader,
        }

        file, err := mapper.ConvertCreateFileIntoFile(createFileDto, bucketName: "test")
        if err != nil { render.JSON(w, r, dto.CreateError( msg: "error")) }

        if err := fh.minioStore.SaveFile(fh.ctx, file); err != nil {
            render.JSON(w, r, dto.CreateError(err.Error()))
            return
        }
        render.JSON(w, r, file)
        w.WriteHeader(http.StatusCreated)
    }
}
```

Рис. 6. Обработчик запросов на сохранение файла

Если запрос прошел все проверки, производится запрос к базе данных на сохранение необходимой метаинформации о файле в таблицу *Files* и таблицу *Folders*, после успешного запроса к базе данных отправляется запрос на сохранение самого файла в хранилище *MinIO* (см. рис. 7). Запросы к базе данных (см. рис. 8) и хранилищу производятся в распределенной транзакции, и в случае возникновения ошибок в одном из запросов транзакция откатывается, за счёт чего сохраняется консистентность данных, и пользователь получает читаемую ошибку в формате *JSON* статус код 500.

```go
func (m *Minio) SaveFile(ctx context.Context, file *model.File) error { 1 usage  ± i.bragin
    reqCtx, cancel := context.WithTimeout(ctx, 10*time.Second)
    defer cancel()

    if _, err := m.minioClient.PutObject(reqCtx, file.BucketName, file.FileId, bytes.NewBuffer(file.Content), file.Size,
        minio.PutObjectOptions{
            ContentType: file.MimeType,
            UserMetadata: map[string]string{
                "Name": file.FileName,
            },
        }); err != nil { return err }
    return nil
}
```

Рис. 7. Функция сохранения файла в объектном хранилище

```go
func (s *Store) SaveFile(ctx context.Context, f model.File) error {  no usages  new *
    _, err := s.db.ExecContext(ctx,  query: `
    INSERT INTO files(file_id, name, path, size, mimetype)
    VALUES ($1, $2, $3, $4, $5);`,
        f.FileId, f.FileName, f.FilePath, f.Size, f.MimeType)
    if err != nil { return err }
    return nil
}
```

Рис. 8. Сохранение метаинформации о файле в базе данных

Еще один метод, представленный в данной группе эндпоинтов, служит для получения файла и метаинформации о нем по *id* файла в системе (см. рис. 9). В методе *GET/ GetFile* проверяется наличие в запросе *id* файла, если параметр отсутствует в строке запроса, то пользователь получит статус-код 400 и читаемое описание ошибки в формате *JSON*. В случае успешной проверки, формируется запрос в объектное хранилище (см. рис. 10). Если валидация пройдена успешно, пользователь получит необходимый файл.

```
func (fh *Handler) GetFile() http.HandlerFunc {  1 usage   i.bragin *
    return func(w http.ResponseWriter, r *http.Request) {
        const op = "GetFile Handler"

        log := fh.log.With(
            slog.String( key: "OP", op),
            slog.String( key: "request_id", middleware.GetReqID(r.Context())),
        )

        fileId := chi.URLParam(r,  key: "id")
        if fileId == "" {
            log.Error( msg: "error: fileId is empty")
            render.JSON(w, r, dto.CreateError( msg: "FileId is empty"))
            return
        }

        file, err := fh.minioStore.GetFile(fh.ctx,  bucketName: "test", fileId)
        if err != nil {
            render.JSON(w, r, dto.CreateError(err.Error()))
            return
        }
        //render.JSON(w, r, mapper.ConvertFileToGetFileDto(file))
        w.Header().Set( key: "Content-Type", file.MimeType)
        w.Write(file.Content)
    }
}
```

Рис. 9. Обработчик запросов на получение

```
func (m *Minio) GetFile(ctx context.Context, bucketName string, fileId string) (*model.File, error) {  1 usage   i.bragin
    reqCtx, cancel := context.WithTimeout(ctx, 5*time.Second)
    defer cancel()
    obj, err := m.minioClient.GetObject(reqCtx, bucketName, fileId, minio.GetObjectOptions{})
    defer obj.Close()
    if err != nil { return nil, err }
    objInfo, err := obj.Stat()
    buffer := make([]byte, objInfo.Size)
    _, err = obj.Read(buffer)
    if err != nil && err != io.EOF { return nil, fmt.Errorf( format: "failed to get objects. err: %w", err) }

    return &model.File{
        FileId:   objInfo.Key,
        FileName: objInfo.UserMetadata["Name"],
        Content:  buffer,
        MimeType: objInfo.ContentType,
        Size:     objInfo.Size,
    }, nil
}
```

Рис. 10. Функция получения файла из объектного хранилища

Для безопасного общения между клиентом и сервером была реализована авторизация и аутентификация с помощью *JWT* (*JSON Web Token*) [5], так как данный способ простой в реализации, и в то же время обеспечивает необходимую безопасность доступа к данным.

На рисунке 11 изображен метод проверки токена, вызываемый при обращении на каждый защищенный эндпоинт. В методе расшифровывается токен, тем самым проверяется его валидность. Так же из токена достается id пользователя и выполняется запрос в базу данных для проверки на наличие такого пользователя в системе. В случае неудачной проверки токена, клиенту возвращается *http* статус код 403 с ошибкой «*Permission denied*».

```go
func WithJWTAuth(handlerFunc http.HandlerFunc, cfg *config.Config, user model.UserStore, ctx context.Context) http.HandlerFunc {  no usages  ivan
    return func(w http.ResponseWriter, r *http.Request) {
        tokenString := validate.GetTokenFromRequest(r)

        token, err := validateJWT(tokenString, cfg)
        if err != nil {
            log.Printf( format: "failed to validate token: %v", err)
            permissionDenied(w)
            return
        }

        if !token.Valid {
            log.Println( v...: "invalid token")
            permissionDenied(w)
            return
        }

        claims := token.Claims.(jwt.MapClaims)
        str := claims["userID"].(string)

        userID, err := strconv.Atoi(str)
        if err != nil {
            log.Printf( format: "failed to convert userID to int: %v", err)
            permissionDenied(w)
            return
        }
        ctx := context.WithValue(r.Context(), UserKey, user)
        u, err := user.GetUserById(ctx, userID)
        if err != nil {
            log.Printf( format: "failed to get user by id: %v", err)
            permissionDenied(w)
            return
        }

        ctx = context.WithValue(ctx, UserKey, u.Id)
        r = r.WithContext(ctx)
        handlerFunc(w, r)
```

Рис. 11. Метод проверки JWT токена

Приложение было развернуто в контейнерах *Docker*, так как это позволяет существенно упростить процесс развертывания на сервере, а также улучшить его переносимость и масштабируемость. Система была развернута в 2-х контейнерах (см. рис. 12-13).

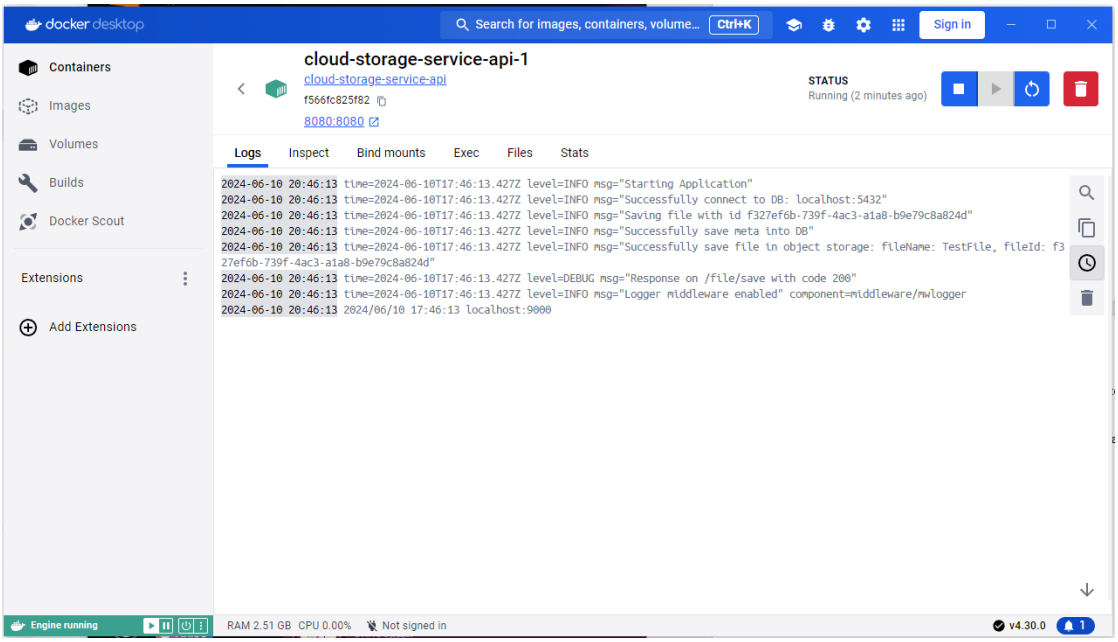

Рис. 12. Приложение, развернутое в контейнере

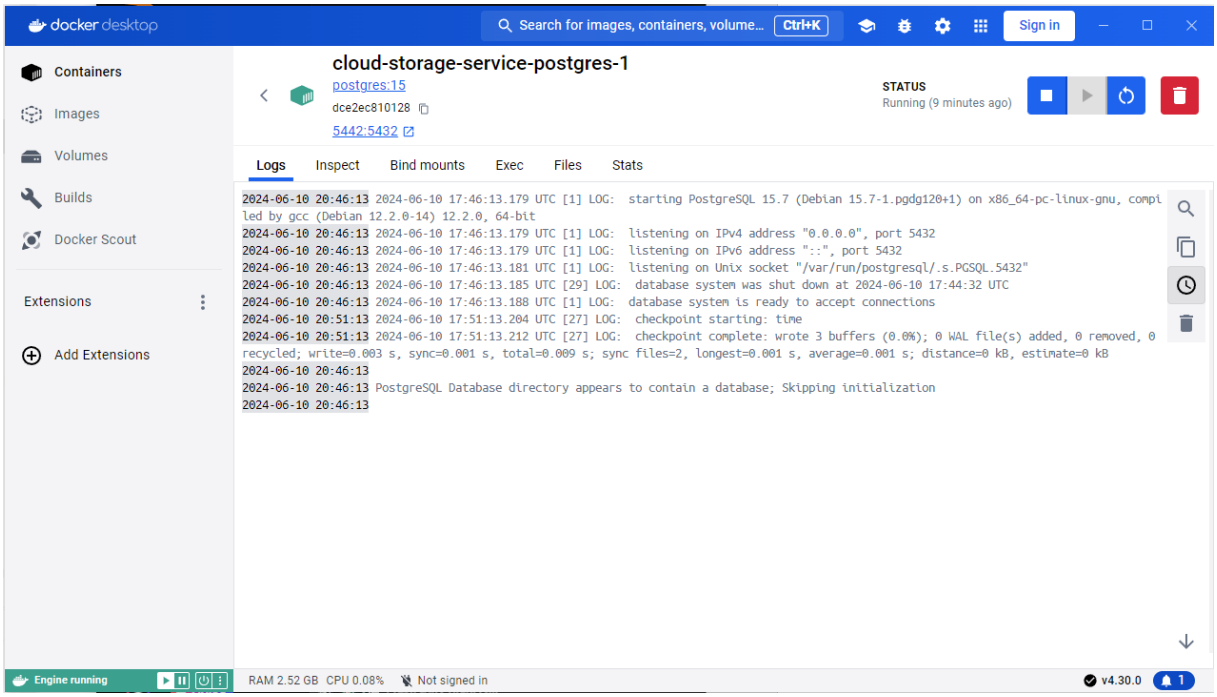

Рис. 13. База данных, развернутая в контейнере

5. Технологии реализации

Для реализации описанного решения были использованы следующие программные средства:

Go – компилируемый многопоточный язык программирования, разработанный внутри компании *Google* [6].

PostgreSQL – свободная объектно-реляционная система управления базами данных (СУБД) [7].

MinIO – высокопроизводительная система хранения объектов, выпущенная под *GNU Affero General Public License версии v*3.0 [8].

Docker – программное обеспечение для автоматизации развёртывания и управления приложениями в средах с поддержкой контейнеризации, контейнеризатор приложений [9].

6. Внедрение и тестирование

Тестирование программного обеспечения является важным этапом в процессе разработки, направленным на выявление дефектов, проверку функциональности и обеспечение качества создаваемого продукта. В рамках данной работы были проведены два вида тестирования для проверки корректности работы системы.

Для проверки отдельных модулей системы были написаны *unit*-тесты (см. рис. 14) с использованием стандартной библиотеки, представленной в языке *golang*.

```go
func TestUserServiceHandlers(t *testing.T) {  Tiago Taquelim
    userStore := &mockUserStore{}
    handler := NewHandler(userStore)

    t.Run( name: "should fail if the user ID is not a number", func(t *testing.T) {
        req, err := http.NewRequest(http.MethodGet,  url: "/user/abc",  body: nil)
        if err != nil { t.Fatal(err) }

        rr := httptest.NewRecorder()
        router := mux.NewRouter()

        router.HandleFunc( path: "/user/{userID}", handler.handleGetUser).Methods(http.MethodGet)

        router.ServeHTTP(rr, req)

        if rr.Code != http.StatusBadRequest {
            t.Errorf( format: "expected status code %d, got %d", http.StatusBadRequest, rr.Code)
        }
    })

    t.Run( name: "should handle get user by ID", func(t *testing.T) {
        req, err := http.NewRequest(http.MethodGet,  url: "/user/42",  body: nil)
        if err != nil { t.Fatal(err) }

        rr := httptest.NewRecorder()
        router := mux.NewRouter()

        router.HandleFunc( path: "/user/{userID}", handler.handleGetUser).Methods(http.MethodGet)

        router.ServeHTTP(rr, req)

        if rr.Code != http.StatusOK {
            t.Errorf( format: "expected status code %d, got %d", http.StatusOK, rr.Code)
        }
    })
}
```

Рис. 14. Модульное тестирование

Так же было проведено ручное тестирование для проверки всех функций системы (см. рис 15). Для отправки запросов на серверную часть в качестве клиента была выбрана программа *Postman*, так как она имеет удобный интерфейс и все необходимые функции, ускоряющие процесс тестирования.

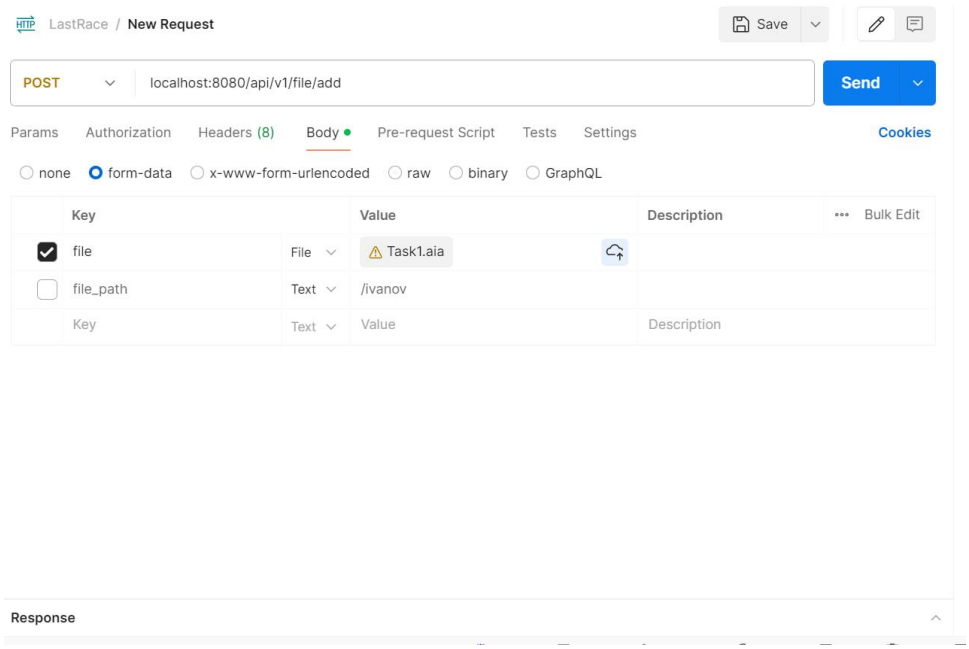

Рис. 15. Отправка запросов в Postman

Заключение

В ходе данной работы было разработано, реализовано раернуто веб-приложение облачного хранилища файлов с целью создания удобного и функционального инструмента для хранения и управления файлами в облаке.

Работа состояла из следующих этапов выполнения:

Проведенный анализ существующих облачных хранилищ позволил выявить их основные особенности и недостатки, что послужило основой для определения требований к разрабатываемому приложению.

Была спроектирована и реализована архитектура приложения, выбраны необходимые технологии и инструменты разработки.

Система прошла тестирование, что гарантирует её высокую производительность и устойчивость к нагрузкам.

Полученное веб-приложение позволяет пользователям загружать, хранить и организовывать файлы в облаке, обеспечивая удобный доступ к данным из любой точки мира. Внедрение дополнительных функций, таких как управление правами доступа, возможность создания пространства для группового хранения файлов, значительно расширяет функциональные возможности приложения.

Список источников

1. Смыгин К. Облачный обзор // Medium [платформа для социальной журналистики]. – Дата публикации: 05.04.2014. –URL: https://medium.com/@kassandr19/-55566e1a97db – Дата обращения: (29.11.2023)

2 What are the best self-hosted cloud storage solutions? — URL: https://www.slant.co/topics/4505/~self-hosted-cloud-storage-solutions (дата обращения: 10.04.2024)

3. Надежная облачная служба хранения и обмена файлами Seafile (обзор) / Moyens Staff. –Журнал Moyens I/O, 2023. – URL: https://ru.moyens.net/обзоры-продуктов/122097/надежная-облачная-служба-хранения-и-о/ (дата обращения: 29.11.2023).

4. Бураков А. REST, что же ты такое? // Школа системного анализа и проектирования Systems.Education. Школа системного анализа и проектирования, 2011—2024. – URL: https://systems.education/what-is-rest (дата обращения: 16.04.2024).

5. JWT.IO - JSON Web Token Introduction. – URL: https://jwt.io/introduction JWT (дата обращения: 16.04.2024).

6. The Go Programming Language – URL: https://go.dev/ (дата обращения: 14.04.2024)

7. PostgreSQL: The world's most advanced open source database –The PostgreSQL Global Development Group, 1996-2024. – URL: https://www.postgresql.org/ (дата обращения: 14.04.2024).

8. MinIO. – MinIO, Inc., 2014-2025. – URL: https://min.io/ (дата обращения: 14.04.2024).

9. Моуэт Э. Использование Docker / пер. с англ. А. В. Снастина; науч. ред. А. А. Маркелов / Э. Моуэт. – Москва : ДМК Пресс, 2017. – 354 с.