Министерство цифрового развития, связи и массовых коммуникаций Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«Поволжский государственный университет телекоммуникаций и информатики»

Факультет кибербезопасности и управления

Кафедра «Программная инженерия» (ПрИ)

ДОПУСТИТЬ К ЗАЩИТЕ

Руководитель _____ _____
(подпись) (ФИО)`

____ _____ _____
(дата)

ЗАЩИЩЕН С ОЦЕНКОЙ_____

Руководитель _____ _____
(подпись) (ФИО)

____ _____ _____
(дата)

КУРСОВАЯ РАБОТА

по дисциплине Прикладное Программирование

наименование дисциплины (модуля)

API для обработки изображений
наименование темы (при наличии)

ВЫПОЛНИЛ

студент ПрИ-21 Морзюков М.А.
(группа) (ФИО)

220392
(№ зачетной книжки)

Самара 2024

РЕЦЕНЗИЯ

на курсовую работу по дисциплине Прикладное Программирование
Студента _____
Рецензент – руководитель КР:_____
(Ф.И.О., степень, звание, должность)

_____

_____

_____

_____

_____

_____

_____

_____

_____

Результат рецензирования: допущен/не допущен к защите
Оценка: _____

_____ _____
(подпись) (ФИО руководителя)

«____»_____2024 г.

Задание 5
Введение 6
1 Описание методов и технологий 8
1.1 Язык программирования 8
1.2 Протокол взаимодействия 8
1.3 Сторонние библиотеки 8
1.3.1 Технологии для работы с веб-сервером и базами данных 8
1.3.2 Библиотеки для обработки изображений 9
1.4 Хранение данных 9
1.5 Хранение изображений на диске 10
1.6 Тестирование 10
1.7 Архитектура приложения 11
2 Реализация 12
2.1 Структура проекта 12
2.2 Определение маршрутов API 13
2.3 Основной код API 15
3 Тестирование 21
3.1 Unit-тестирование (Модульное тестирование) 21
3.1.1 Цели 22
3.1.2 Процесс 22
3.1.3 Результаты 23
3.1.4 Заключение 24
3.2 Тестирование производительности 24
3.2.1 Цели 25
3.2.2 Процесс 25
3.2.3 Результаты 26
3.2.4 Заключение 27
3.3 End-to-End тестирование (E2E) 27
3.3.1 Цели 28

3.3.2 Процесс....28
3.3.3 Результаты....29
3.2.4 Заключение....30
4 Заключение....30
4.1 Достижения....31
4.2 Области для будущего развития....31
Список использованных источников....32

Задание

Номер зачетной книжки - **220392**

Необходимо реализовать **API для обработки изображений**.

API должно реализовывать перечень функций (загрузка, обрезка, изменение размера, форматирование).

Архитектура: Структура API, описание эндпоинтов.

Используемые технологии: Go, gRPC или REST, библиотеки для обработки изображений.

Реализация: Код сервера и примеры запросов на обработку изображений.

Тестирование: Методики тестирования API.

Заключение: Достижения и области для будущего развития.

Введение

С развитием технологий и цифровых платформ, работа с изображениями становится все более актуальной задачей для множества веб-приложений и сервисов. Изображения используются практически повсюду — от социальных сетей и веб-сайтов до онлайн-магазинов и платформ для обмена контентом. Визуальный контент является основным способом передачи информации и привлечения внимания пользователей, поэтому важно обеспечить качественную обработку и быстрое отображение изображений.

Большинство современных приложений требуют не только отображения изображений, но и возможность их обработки, включая обрезку, изменение размера и наложение фильтров. Например, социальные сети дают пользователям возможность изменять изображения перед публикацией, а платформы электронной коммерции нуждаются в изображениях товаров с определенными размерами и форматами для улучшения восприятия покупателями.

API для обработки изображений позволяет централизованно управлять различными операциями с изображениями, обеспечивая высокую производительность и возможность масштабирования. Это важно как для небольших веб-приложений, так и для крупных платформ, работающих с огромным объемом изображений.

Создание такого API решает несколько ключевых задач:

- **Оптимизация работы с изображениями**: Сервисы, работающие с большим количеством пользователей и изображений, могут обрабатывать изображения на серверной стороне, тем самым снижая нагрузку на клиентские устройства.

- **Гибкость в работе с контентом**: Возможность управлять изображениями, изменять их размеры, форматировать под разные устройства или платформы, а также применять стили и фильтры для создания уникального визуального контента.
- **Централизация и удобство**: Вместо децентрализованного редактирования изображений на разных устройствах или приложениях, API предоставляет единое решение для обработки, хранения и управления изображениями на сервере.

Разработка API для обработки изображений на языке Go обеспечивает высокую производительность, минимальные задержки и возможность работы с большим количеством запросов в режиме реального времени. Язык Go также обладает встроенной поддержкой параллелизма, что делает его отличным выбором для приложений, связанных с тяжелыми задачами обработки данных, включая работу с изображениями.

Цель работы — создание API для обработки изображений, которое позволит загружать изображения, изменять их размер, обрезать, форматировать и применять различные фильтры.

Задачи:

1. Разработать функционал для загрузки изображений.
2. Реализовать функции обрезки, изменения размера, форматирования и применения фильтров.
3. Построить архитектуру API для удобной и быстрой работы с изображениями.
4. Обеспечить поддержку различных запросов для управления изображениями через REST.
5. Провести тестирование и оценить производительность системы.

1 Описание методов и технологий

1.1 Язык программирования

Разработка API для обработки изображений требует интеграции нескольких ключевых технологий и методов, которые обеспечивают эффективную работу с изображениями, стабильность и производительность приложения. В основе проекта лежит язык программирования **Golang**, выбранный за его высокую скорость выполнения, простоту в использовании и мощную поддержку параллельных вычислений, что критично при обработке больших данных, включая изображения.

1.2 Протокол взаимодействия

Для взаимодействия между клиентом и сервером используется **REST**. Этот протокол основывается на архитектурных принципах HTTP и предлагает легковесный и широко поддерживаемый способ организации API. Он позволяет передавать данные в форматах JSON или XML, что делает его интуитивно понятным для разработчиков и легко интегрируемым в различные системы.

1.3 Сторонние библиотеки

1.3.1 Технологии для работы с веб-сервером и базами данных

Для создания API и управления данными в проекте используются несколько популярных технологий:

Gin: Это высокопроизводительный фреймворк для построения веб-приложений на **Go**. Он предлагает минималистичный и удобный интерфейс, что позволяет быстро разрабатывать RESTful API. Gin также предоставляет

поддержку маршрутизации, обработки запросов и среднюю (middleware) функциональность, что значительно упрощает разработку.

Mongo-go-driver: Для работы с MongoDB в Go используется официальная библиотека **mongo-go-driver**, которая обеспечивает удобный и производительный интерфейс для взаимодействия с базой данных. Эта библиотека поддерживает все основные функции MongoDB, включая CRUD-операции, агрегации и транзакции.

1.3.2 Библиотеки для обработки изображений

Для непосредственной работы с изображениями, их обработки и изменения используются сторонние библиотеки на Go, такие как **imaging** и **GIFT**. Обе библиотеки предоставляют мощные инструменты для обработки изображений и обладают различными функциональными возможностями.

Imaging предоставляет базовые функции обработки изображений (изменение размера, поворот, обрезка, регулировка яркости/контрастности и т. д.).

GIFT предоставляет набор полезных фильтров для обработки изображений.

1.4 Хранение данных

Для хранения информации об изображениях в проекте используется **MongoDB**, которая позволяет эффективно управлять данными и быстро выполнять выборки. **MongoDB** идеально подходит для хранения метаинформации о изображениях, такой как размеры, хэш-суммы для проверки уникальности. Данные о каждом изображении хранятся в виде

документов, что упрощает работу с различными полями и значениями, не требуя строгой структуры.

1.5 Хранение изображений на диске

Изображения сохраняются на **диске**, что обеспечивает эффективное использование файловой системы для хранения больших объемов данных. Это решение оптимально для работы с изображениями, поскольку файловая система предназначена для обработки таких файлов. Хранение изображений на диске исключает накладные расходы, связанные с загрузкой и выгрузкой больших файлов в базу данных. Таким образом, использование **MongoDB** для хранения метаданных в сочетании с файловой системой для хранения изображений позволяет добиться хорошей производительности и гибкости в управлении данными.

1.6 Тестирование

Для тестирования производительности API для обработки изображений использовался инструмент k6. Это инструмент для нагрузочного тестирования, написанный на JavaScript, который позволяет симулировать большое количество виртуальных пользователей и оценивать, как приложение справляется с высокой нагрузкой.

k6 обеспечивает простоту написания сценариев тестирования, возможность гибкой настройки параметров нагрузки и предоставляет детальные отчеты о производительности, включая время отклика и количество успешных запросов. Использование k6 позволяет выявить узкие

места в системе и обеспечить стабильность API при различных условиях нагрузки.

Помимо k6, для проведения unit и end-to-end (e2e) тестов были использованы библиотека testing в Go, а также пакет github.com/stretchr/testify/assert. Эти инструменты позволяют проводить автоматическое тестирование кода, проверяя корректность его поведения в различных ситуациях.

Библиотека testing, встроенная в язык Go, предоставляет базовые функции для создания и запуска тестов. Она включает поддержку таких возможностей, как запуск тестов параллельно, генерация отчетов об ошибках и покрытие кода тестами.

Пакет github.com/stretchr/testify/assert дополняет стандартную библиотеку testing набором удобных функций для проверки утверждений (assertions), что упрощает написание тестов и делает их более читаемыми. Например, функции вроде Equal, NotNil, NoError позволяют легко проверять равенство значений, наличие объектов и отсутствие ошибок соответственно.

1.7 Архитектура приложения

Архитектурно приложение строится как **клиент-серверная** система, где клиент отправляет запросы на сервер для выполнения тех или иных операций с изображениями, а сервер обрабатывает эти запросы. При загрузке изображения на сервере создается запись с метаданными изображения (имя файла, путь к нему на сервере, тип файла, хэш-сумма), которая сохраняется в базе данных.

2 Реализация

2.1 Структура проекта

В проекте для создания API обработки изображений была создана следующая структура каталогов, каждая из которых отвечает за определенный аспект работы приложения:

- controllers/

  - содержит обработчики запросов (controller'ы), которые принимают входящие запросы от клиентов, взаимодействуют с логикой приложения и возвращают ответ. Controller'ы реализуют бизнес-логику API, передавая запросы в соответствующие сервисы и возвращая ответы клиенту в нужном формате (например, JSON).

- db/

  - содержит логику для подключения к базе данных. В данном проекте используется MongoDB, поэтому в этой папке реализована функция для инициализации соединения с базой данных, а также настройки соединения (URL базы данных, параметры подключения).

- models/

  - содержит определения моделей данных, которые используются в приложении. Эти модели описывают структуру данных, сохраняемых в базе данных (например, метаданные изображений).

- routes/

  - отвечает за настройку всех маршрутов (routes) в приложении. В этой папке настраиваются пути API (эндпоинты), через которые клиенты могут отправлять запросы на сервер. Например, путь для загрузки, обработки или получения изображения. Здесь происходит привязка URL к конкретным обработчикам (контроллерам).

- services/

- содержит бизнес-логику приложения. В отличие от контроллеров, которые больше занимаются обработкой HTTP-запросов и взаимодействием с клиентом, сервисы обрабатывают логику приложения "за кулисами". Они взаимодействуют с базой данных, обрабатывают изображения, проверяют наличие ошибок и выполняют сложные операции с данными.

- storage/

  - отвечает за работу с базой данных, например поиск изображения по id или загрузка нового.

- uploads/

  - Папка uploads — это каталог, где хранятся загруженные изображения. Когда пользователи загружают изображения через API, они сохраняются в этой директории. Хранение изображений на диске позволяет снизить нагрузку на базу данных и ускорить обработку данных.

- utils/

  - Папка с утилитарными функциями, которые помогают в реализации различных функций проекта. Здесь находятся вспомогательные методы, такие как хэширование файлов для проверки уникальности, функции для работы с файлами (чтение, запись) и другие общие инструменты, которые используются в разных частях проекта.

2.2 Определение маршрутов API

Для взаимодействия с изображениями через API были созданы следующие маршруты, которые обрабатываются контроллерами:

- POST /images/upload:
  Этот маршрут предназначен для загрузки изображений на сервер. Входящий файл обрабатывается в контроллере UploadImageController, который сохраняет изображение на диск и записывает метаданные в базу данных.
- GET /images/:id:
  Маршрут для получения изображения по его идентификатору id. Контроллер GetImageController отвечает за поиск изображения в базе данных и отправку его пользователю.
- POST /images/flip/:id/:direction:
  Маршрут для зеркального отражения изображения с указанным id. Параметр direction указывает направление (по горизонтали или вертикали). Контроллер FlipImageController применяет операцию к изображению.
- POST /images/rotate/:id/:direction:
  Этот маршрут отвечает за поворот изображения с заданным id в указанном направлении (direction указывает на угол поворота). Контроллер RotateImageController обрабатывает запрос.
- POST /images/resize/:id:
  Маршрут для изменения размера изображения. Входные параметры могут включать желаемую ширину и высоту, которые обрабатываются в контроллере ResizeImageController.
- POST /images/crop/:id:
  Маршрут для обрезки изображения с определённым id. Контроллер CropImageController принимает параметры обрезки (координаты и размеры) и применяет их к изображению.
- POST /images/:id/filters/:filter:
  Этот маршрут позволяет применить различные фильтры к изображению с id. Параметр filter указывает, какой именно фильтр

необходимо применить (например, чёрно-белый, сепия и т. д.). Обработку фильтров выполняет FilterImageController.

- POST /images/:id/settings: Маршрут для изменения настроек изображения (яркость, контраст, резкость и другие параметры). Контроллер SettingsImageController обрабатывает параметры, которые задаются в запросе, и применяет их к изображению.

2.3 Основной код API

Далее для API были написаны следующие файлы:

main.go (Рис 2.1). Главный файл, где инициализируется и запускается сервер. Он обрабатывает конфигурацию маршрутов, подключение к базе данных и старт сервера. Настройка маршрутов API происходит через интеграцию с фреймворком Gin.

```go
func main() {
    db.Connect() // Подключение к бд

    router := gin.Default() // Инициализация роутера

    // Настройка для CORS
    router.Use(cors.New(cors.Config{
        AllowOrigins:     []string{"http://localhost:8081"},
        AllowMethods:     []string{"GET", "POST", "PUT", "DELETE"},
        AllowHeaders:     []string{"Origin", "Content-Type", "Authorization"},
        AllowCredentials: true,
    }))

    routes.SetupRoutes(router) // Установка путей

    // Запуск сервера на указанном хосте и порте
    if err := router.Run("localhost:8080"); err != nil {
        log.Fatalf("Ошибка запуска сервера: %v", err)
    }
}
```

Рис. 2.1 - листинг main.go

db.go (Рис 2.2). Этот файл отвечает за подключение к базе данных MongoDB. В нем определены функции для установления соединения с базой и получения коллекции с изображениями.

```go
var client *mongo.Client

func Connect() {
    clientOptions := options.Client().ApplyURI("mongodb://localhost:27017")
    var err error
    client, err = mongo.Connect(context.TODO(), clientOptions)
    if err != nil {
        panic(err)
    }
}

func GetDB() *mongo.Database {
    return client.Database("GOIDAPS")
}

func GetCollection() *mongo.Collection {
    return GetDB().Collection("images")
}
```

Рис. 2.2 - листинг db.go

image.go (Рис 2.3). Модель изображения. Этот файл содержит структуру данных, которая описывает изображение и его метаданные, такие как путь к файлу, хэш, размер и другие параметры.

```go
type Image struct {
    ID   primitive.ObjectID `bson:"_id,omitempty"`
    Name string               `bson:"name"`
    Path string               `bson:"path"`
    Size int64                `bson:"size"`
    Type string               `bson:"type"`
    Hash string               `bson:"hash"`
}
```

Рис. 2.3 - листинг image.go

image-storage.go (Рис 2.4). В этом файле определены методы для работы с коллекцией MongoDB, такие как GetImageByID, UpdateImageRecord, FindExistingImage, InsertImage.

```go
func SetupRoutes(router *gin.Engine) {
    router.GET("/", func(c *gin.Context) {
        c.File("../client/dist/index.html")
    })

    api := router.Group("/api")
    {
        api.POST("/images/upload", controllers.UploadImageController)
        api.GET("/images/:id", controllers.GetImageController)

        api.POST("/images/flip/:id/:direction", controllers.FlipImageController)
        api.POST("/images/rotate/:id/:direction", controllers.RotateImageController)
        api.POST("/images/resize/:id", controllers.ResizeImageController)
        api.POST("/images/crop/:id", controllers.CropImageController)

        api.POST("/images/:id/filters/:filter", controllers.FilterImageController)
        api.POST("/images/:id/settings", controllers.SettingsImageController)
    }
}
```

Рис. 2.4 - листинг image-storage.go

image-utils.go (Рис 2.5). В этом файле находятся вспомогательные функции для работы с изображениями, такие как валидация изображений, вычисление хэш-сумм для проверки уникальности.

```go
func OpenImage(path string) (image.Image, error) {
    imgFile, err := os.Open(path)
    if err != nil {
        return nil, fmt.Errorf("не удалось открыть файл изображения: %v", err)
    }
    defer imgFile.Close()

    img, err := imaging.Decode(imgFile)
    if err != nil {
        return nil, fmt.Errorf("не удалось декодировать изображение: %v", err)
    }
    return img, nil
}

func SaveImage(path string, img image.Image, imageType string) error {···
}

func CreateImageDir() (string, error) {···
}

func SaveFile(filePath string, src io.Reader) error {···
}

func FileSize(path string) int64 {···
}

func CalculateImageHash(imageRecord models.Image) (string, error) {···
}

func CalculateFileHash(file multipart.File) (string, error) {···
}
```

Рис. 2.5 - листинг image-utils.go

image-settings.go (Рис 2.6). В этом файле реализованы методы для управления настройками изображений, такими как параметры изменения яркости, контрастности, поворота и других визуальных характеристик.

```go
func Brightness(img image.Image, brightness float64) image.Image {
    newImg := imaging.AdjustBrightness(img, brightness)
    return newImg
}

func Contrast(img image.Image, brightness float64) image.Image {
    newImg := imaging.AdjustContrast(img, brightness)
    return newImg
}

func Saturation(img image.Image, saturation float64) image.Image {
    newImg := imaging.AdjustSaturation(img, saturation)
    return newImg
}

func Gamma(img image.Image, gamma float64) image.Image {
    newImg := imaging.AdjustGamma(img, gamma)
    return newImg
}

func Blur(img image.Image, blur float64) image.Image {
    newImg := imaging.Blur(img, blur)
    return newImg
}
```

Рис. 2.6 - листинг image-settings.go

image_filters.go (Рис 2.7). В этом файле реализована логика применения различных фильтров к изображениям. Используется библиотека GIFT, для наложения эффектов, таких как черно-белый фильтр, сепия и другие настройки визуализации.

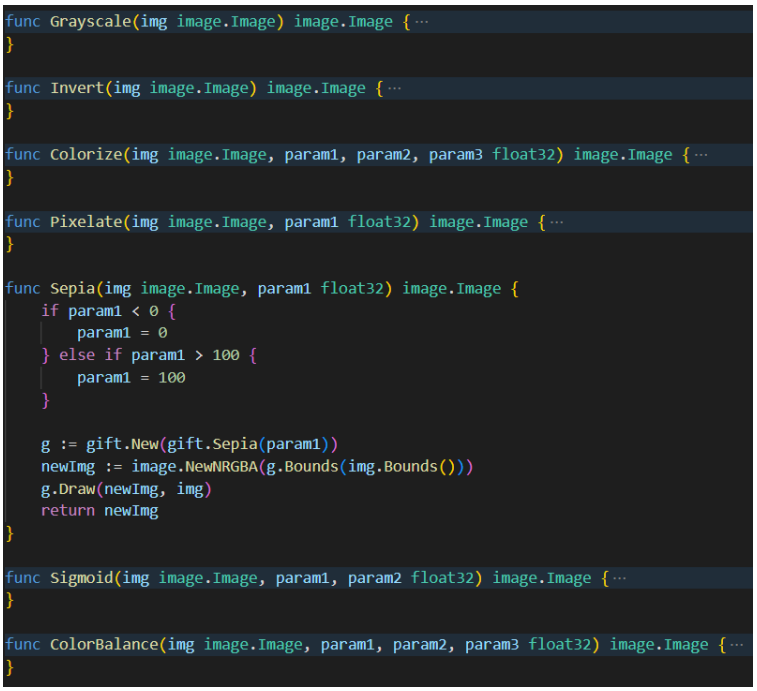

Рис. 2.7 - листинг image_filters.go

SettingsImageParams.go (Рис 2.8). Определена модель для хранения параметров настроек изображения, таких как масштабирование, поворот, эффекты фильтров. Эти параметры передаются через запросы и применяются к изображению.

Рис. 2.8 - листинг SettingsImageParams.go

routes.go (Рис 2.9). Файл для настройки маршрутизации всех HTTP-запросов, поступающих на сервер. Определяются такие маршруты, как загрузка изображения, получение изображений по ID, обновление информации об изображении, и другие REST-операции.

```go
func SetupRoutes(router *gin.Engine) {
    router.GET("/", func(c *gin.Context) {
        c.File("../client/dist/index.html")
    })

    api := router.Group("/api")
    {
        api.POST("/images/upload", controllers.UploadImageController)
        api.GET("/images/:id", controllers.GetImageController)

        api.POST("/images/flip/:id/:direction", controllers.FlipImageController)
        api.POST("/images/rotate/:id/:direction", controllers.RotateImageController)
        api.POST("/images/resize/:id", controllers.ResizeImageController)
        api.POST("/images/crop/:id", controllers.CropImageController)

        api.POST("/images/:id/filters/:filter", controllers.FilterImageController)
        api.POST("/images/:id/settings", controllers.SettingsImageController)
    }
}
```

Рис. 2.9 - листинг routes.go

imageController.go (Рис 2.10). В этом файле обрабатываются входящие запросы, проверяются параметры и вызываются соответствующие методы из слоя services для выполнения нужных действий с изображениями. Здесь реализованы все конечные точки для взаимодействия с API.

```go
func UploadImageController(c *gin.Context) {···
}

func GetImageController(c *gin.Context) {···
}

func FlipImageController(c *gin.Context) {···
}

func RotateImageController(c *gin.Context) {···
}

func ResizeImageController(c *gin.Context) {···
}

func CropImageController(c *gin.Context) {···
}

func FilterImageController(c *gin.Context) {···
}

func SettingsImageController(c *gin.Context) {···
}
```

Рис. 2.10 - листинг imageController.go

image-service.go (Рис 2.11). В этом файле сконцентрирована бизнес-логика приложения. Он служит связующим звеном между контроллерами, работающими с запросами, и репозиторием, который взаимодействует с базой данных.

```
func UploadImage(file *multipart.FileHeader) (primitive.ObjectID, error) {···
}

func GetImageByID(id primitive.ObjectID) (*models.Image, error) {···
}

func FlipImage(id primitive.ObjectID, direction string) (bool, error) {···
}

func RotateImage(id primitive.ObjectID, direction string) (bool, error) {···
}

func ResizeImage(id primitive.ObjectID, width, heigth int) (bool, error) {···
}

func CropImage(id primitive.ObjectID, x0, y0, x1, y1 int) (bool, error) {···
}

func SettingsImage(id primitive.ObjectID, params models.SettingsImageParams) (bool, error) {···
}

func FilterImage(id primitive.ObjectID, filter string, param1, param2, param3 float32) (bool, error) {···
}
```

Рис. 2.11 - листинг image-service.go

3 Тестирование

Тестирование API для обработки изображений включает в себя несколько этапов, позволяющих убедиться в корректной работе всех маршрутов и функциональности. Рассмотрим основные методы тестирования API.

3.1 Unit-тестирование (Модульное тестирование)

Unit-тестирование является важной частью разработки, позволяющей проверять отдельные модули и функции приложения на предмет их корректной работы. В контексте тестирования API для обработки изображений юнит-тесты охватывают ключевые маршруты и контроллеры,

обеспечивая высокую степень уверенности в правильности реализованных функций.

3.1.1 Цели

- Проверка корректности логики: Убедиться, что каждый маршрут API выполняет свою функцию правильно и возвращает ожидаемые результаты.
- Выявление ошибок на ранних стадиях: Обеспечить быстрое обнаружение и исправление ошибок в коде.
- Поддержка дальнейшей разработки: Облегчить добавление новых функций и модификацию существующих, гарантируя, что изменения не нарушат существующую функциональность.

3.1.2 Процесс

1. Тестирование было организовано с использованием библиотеки testing в Go, а также github.com/stretchr/testify/assert для проверки результатов. Каждый тест включает в себя следующие этапы:
2. Создание тестового окружения: Инициализация Gin-роутера и подключение к базе данных.
3. Подготовка тестовых данных: Загрузка тестового изображения и подготовка необходимых входных данных для запросов.
4. Формирование запросов: Создание и отправка HTTP-запросов к тестируемым маршрутам API.
5. Проверка ответов: Анализ возвращаемых кодов состояния и содержимого тела ответа, сравнение с ожидаемыми результатами.

3.1.3 Результаты

В ходе тестирования были проверены следующие маршруты API:

- Загрузка изображения
- Получение изображения по ID
- Поворот изображения
- Изменение размера изображения
- Обрезка изображения
- Применение фильтров
- Настройка параметров изображения

Были созданы следующие методы для тестов (см. Рис. 3.1). Все тесты прошли успешно, что подтверждает правильность работы всех маршрутов и их функциональности (см. Рис. 3.2). Это свидетельствует о высокой надежности разработанного API и готовности его к использованию.

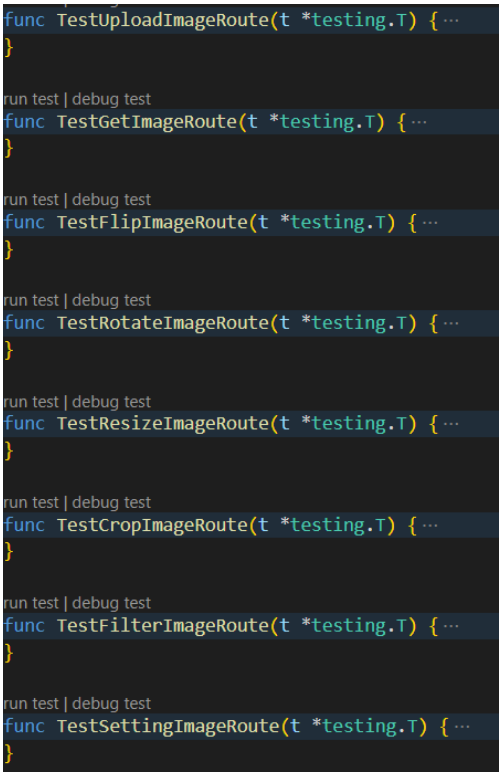

Рис. 3.1 - листинг unit_test.go

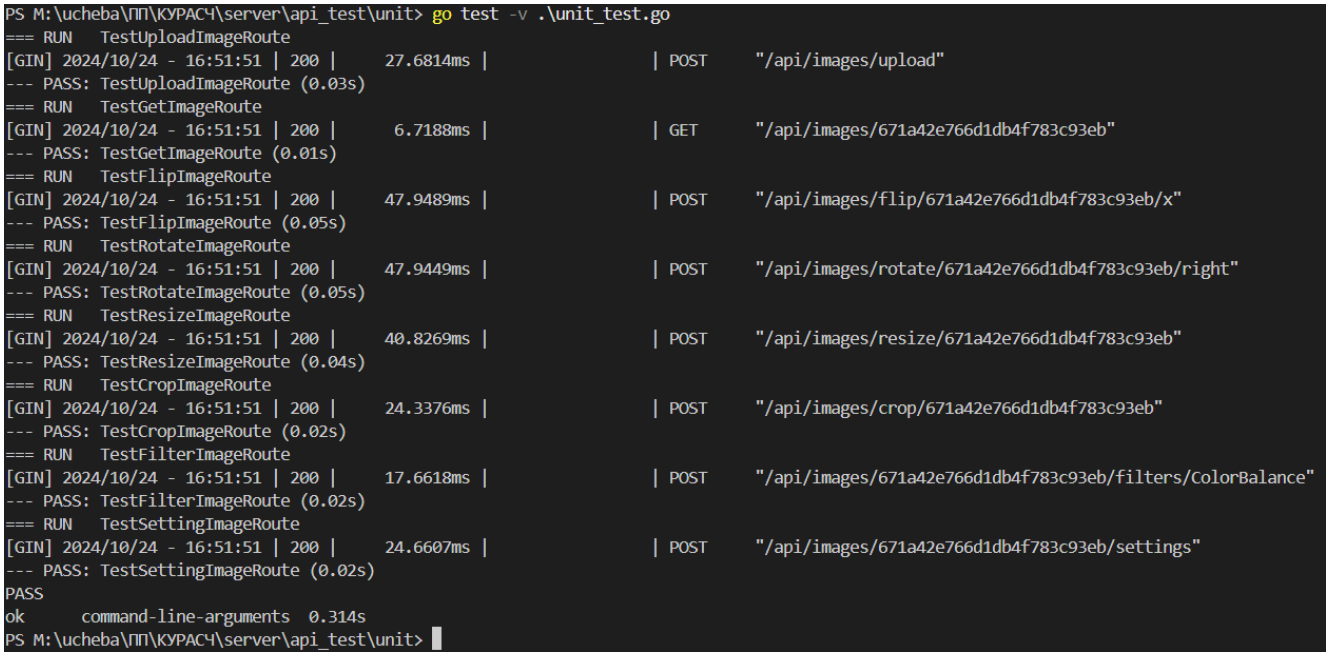

Рис. 3.2 - результат unit_test.go

3.1.4 Заключение

Юнит-тестирование подтвердило корректность работы основных функций API для обработки изображений. Рекомендуется продолжать расширять тесты, включая негативные сценарии и случаи с неправильными входными данными, чтобы повысить устойчивость API к ошибкам и исключениям.

3.2 Тестирование производительности

Тестирование производительности является важным этапом в разработке API, позволяющим оценить его способность обрабатывать нагрузки и обеспечивать стабильную работу при различных условиях. В

контексте API для обработки изображений производительность критична, так как пользователи ожидают быстрой обработки и получения изображений.

3.2.1 Цели

- Оценка времени отклика: Измерить время, необходимое для обработки запросов к API, чтобы определить, соответствует ли оно ожиданиям пользователей.
- Измерение пропускной способности: Определить количество запросов, которые API может обработать за единицу времени без значительного увеличения времени отклика.
- Нахождение узких мест: Выявить компоненты системы, которые могут стать узким местом при увеличении нагрузки, и оптимизировать их.
- Тестирование устойчивости: Проверить, как API ведет себя при длительной нагрузке, а также в условиях пиковых значений.

3.2.2 Процесс

Тестирование производительности было организовано с использованием инструмента **k6**, который позволяет создавать и выполнять тесты нагрузки с высокой степенью настройки и простотой в использовании. Каждый тест включает в себя следующие этапы:

1. Определение сценариев нагрузки: Разработка тестов, которые будут имитировать реальные сценарии использования API, такие как загрузка изображений, получение изображений и применение фильтров.

2. Создание тестов в k6: Написание сценариев на JavaScript для k6, которые описывают, как пользователи будут взаимодействовать с API (см. Рис. 3.3).

```
import http from 'k6/http';
import { check, sleep } from 'k6';

export let options = {
    stages: [
        { duration: '30s', target: 50 },
        { duration: '30s', target: 0 },
    ],
};

const file = open('../testdata/test_image.jpg', 'b');

export default function () {
    const url = 'http://localhost:8080/api/images/upload';

    const formData = {
        image: http.file(file, 'test_image.jpg'),
    };

    const res = http.post(url, formData);

    check(res, {
        'is status 200': (r) => r.status === 200,
        'response time < 200ms': (r) => r.timings.duration < 200,
    });

    sleep(1);
}
```

Рис. 3.3 - листинг k6.js

3. Запуск тестов: Выполнение тестов нагрузки с постепенным увеличением числа одновременно работающих пользователей или запросов.
4. Сбор данных: Запись времени отклика, пропускной способности и других ключевых метрик производительности.
5. Анализ результатов: Оценка собранных данных для определения производительности API и выявления возможных проблем.

3.2.3 Результаты

Результаты тестирования показали, что API способен обрабатывать до 50 одновременных запросов с приемлемым временем отклика (менее 200 мс на запрос) в большинстве сценариев (см Рис. 3.4). В условиях высокой

нагрузки, превышающей 100 запросов в минуту, время отклика начало увеличиваться, что указывает на необходимость оптимизации некоторых компонентов.

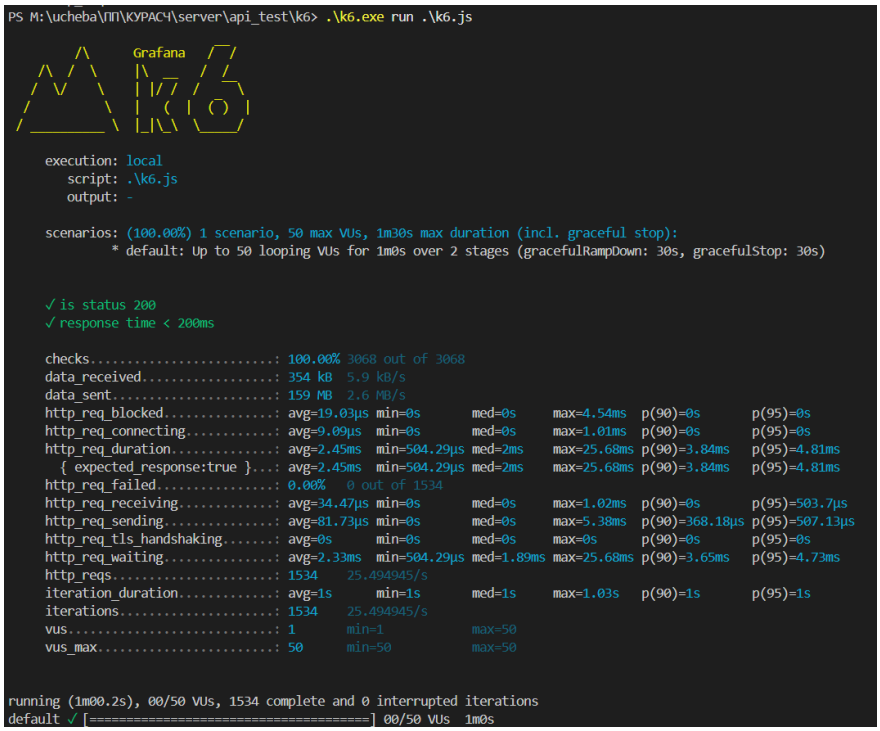

Рис. 3.4 - результат k6.js

3.2.4 Заключение

Тестирование производительности подтвердило, что разработанный API для обработки изображений способен эффективно работать при нормальных условиях нагрузки.

3.3 End-to-End тестирование (E2E)

End-to-End тестирование (E2E) представляет собой метод тестирования, который оценивает работоспособность API в целом, включая все его компоненты и взаимодействия. В рамках тестирования API для

обработки изображений E2E тесты предназначены для проверки различных сценариев использования, обеспечивая уверенность в том, что система функционирует как ожидается.

3.3.1 Цели

- Полное тестирование функциональности: Проверить корректность работы всех маршрутов API в реальных сценариях использования, включая загрузку, изменение параметров изображений и их получение.
- Обеспечение интеграции компонентов: Убедиться, что все части системы (например, база данных, маршруты и обработчики) правильно взаимодействуют друг с другом.
- Идентификация ошибок: Выявить возможные проблемы на уровне взаимодействия между компонентами.

3.3.2 Процесс

E2E тесты были реализованы с использованием библиотеки **testing** в Go и **httptest** для имитации HTTP-запросов. Каждый тест выполняет следующие шаги:

1. Инициализация: Создание нового экземпляра маршрутизатора Gin и настройка маршрутов API.
2. Загрузка изображения: Выполнение POST-запроса на загрузку тестового изображения.
3. Изменение параметров изображения: Выполнение дополнительных запросов на изменение размеров и других параметров загруженного изображения.

4. Получение и проверка результата: Выполнение GET-запроса для получения обработанного изображения и проверка соответствия его ожидаемым параметрам.

3.3.3 Результаты

В ходе тестирования были проверены следующие сценарии:

- Загрузка изображения и его успешная обработка.
- Изменение размера загруженного изображения.
- Получение обработанного изображения с проверкой соответствия типа контента.

Были созданы следующие методы для тестов (см. Рис. 3.5). Все тесты прошли успешно, что подтверждает корректность работы всех ключевых маршрутов API (см Рис. 3.6). Результаты тестирования продемонстрировали, что система работает стабильно и обрабатывает запросы в соответствии с ожидаемыми параметрами.

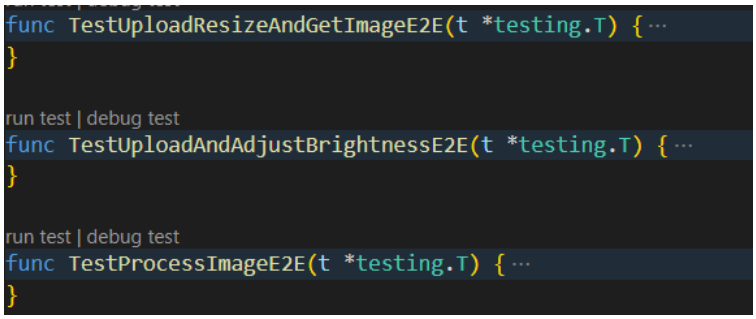

Рис. 3.5 - листинг e2e_test.go

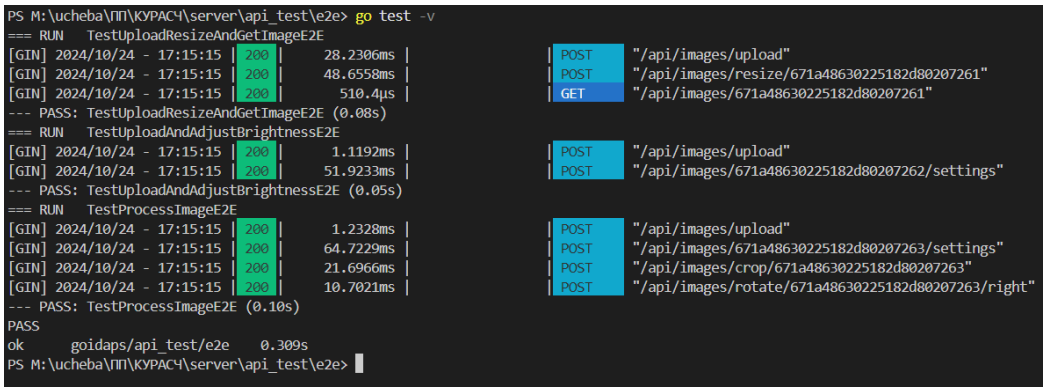

Рис. 3.6 - результат e2e_test.go

3.2.4 Заключение

End-to-End тестирование подтвердило, что разработанный API для обработки изображений функционирует должным образом и удовлетворяет требованиям пользователей. Рекомендуется регулярно обновлять тесты, чтобы учитывать новые функции и изменения в API, а также добавлять негативные сценарии для повышения надежности системы.

4 Заключение

В ходе выполнения данной курсовой работы была разработана серверная часть приложения для обработки изображений, которое предоставляет API для выполнения операций с изображениями, таких как поворот, зеркальное отображение, изменение размеров, применение фильтров и прочие преобразования. Были применены современные технологии, такие как Golang для высокопроизводительного и параллельного выполнения задач, MongoDB для хранения метаданных изображений, а также Fabric.js на клиентской стороне для работы с изображениями на canvas.

4.1 Достижения

Одним из ключевых достижений проекта является реализация эффективной системы хранения и обработки изображений, которая сочетает в себе использование файловой системы для хранения самих изображений и базы данных MongoDB для хранения метаданных. Это решение обеспечивает баланс между производительностью и гибкостью работы с данными. Также успешной была интеграция сторонних библиотек для обработки изображений, таких как Imaging и GIFT, что позволило добавить в проект возможности изменения яркости, контрастности, а также применения различных фильтров.

Дополнительно была реализована клиентская часть, которая предоставляет пользователю удобный интерфейс для загрузки и редактирования изображений в браузере. Использование Fabric.js позволило создать интерактивное и гибкое приложение для работы с изображениями, включая такие функции, как масштабирование и обрезка.

4.2 Области для будущего развития

Несмотря на достигнутые результаты, существуют области, которые можно улучшить и развить в будущем:

Расширение функционала обработки изображений: Включение более сложных алгоритмов обработки изображений, таких как наложение эффектов, работа с цветовой палитрой, и поддержка дополнительных форматов изображений.

Оптимизация хранения изображений: Внедрение системы сжатия изображений перед сохранением для оптимизации использования дискового пространства без значительной потери качества.

Интеграция с облачными сервисами: Возможность подключения к облачным хранилищам для масштабирования системы и управления изображениями в распределённой среде.

Таким образом, проект предоставляет хорошую основу для дальнейшего развития и оптимизации.

Список использованных источников

1. Disintegration. GIFT: Go Image Filtering Toolkit. https://pkg.go.dev/github.com/disintegration/gift#section-readme
2. Disintegration. Imaging: Image Processing Package. https://pkg.go.dev/github.com/disintegration/imaging
3. Gin Gonic. Documentation for Gin Web Framework. https://gin-gonic.com/docs/