

МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования

**«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»**

Кафедра дискретной математики и информационных технологий

**РАЗРАБОТКА ПРИЛОЖЕНИЯ ДЛЯ ОБЛАЧНОГО
ХРАНЕНИЯ С ИСПОЛЬЗОВАНИЕМ РАЗНЫХ СЕРВИСОВ ПО
ПРИНЦИПУ RAID**

БАКАЛАВРСКАЯ РАБОТА

студента 4 курса 421 группы
направления 09.03.01 — Информатика и вычислительная техника
факультета КНиИТ
Захарова Сергея Алексеевича

Научный руководитель
старший преподаватель

П. О. Дмитриев

Заведующий кафедрой
доцент, к. ф.-м. н.

Л. Б. Тяпаев

Саратов 2025

СОДЕРЖАНИЕ

ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ	3
ВВЕДЕНИЕ	4
1 Описание предметной области	5
1.1 Актуальность поставленной задачи	5
1.2 Исследование инструментальных средств разработки	6
1.3 Обзор существующих решений	7
2 Постановка задачи	8
3 Теоретические сведения	10
3.1 Общие сведения о RAID	10
3.2 <i>RAID 5</i>	13
4 Разработка решения для обработки данных на облачных сервисах	16
4.1 Используемые технологии	16
4.2 Яндекс Диск	19
4.3 Dropbox	22
4.4 GoogleDrive	26
5 Реализация класса, распределяющего данные среди облачных хранилищ по принципу RAID 5	31
5.1 Загрузка файла с распределением	33
5.2 Считывание данных	35
6 Разработка пользовательского интерфейса	37
7 Тестирование реализованного решения	47
ЗАКЛЮЧЕНИЕ	50
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	51
Приложение А Исходный код Program	53
Приложение Б Исходный код Startup	56
Приложение В Исходный код YandexDiskService	57
Приложение Г Исходный код DropboxService	61
Приложение Д Исходный код GoogleDriveService	65
Приложение Е Исходный код Raid5Service	71
Приложение Ж Исходный код Raid5Controller	91
Приложение З Исходный код App.vue	95

ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

RAID — это технология объединения нескольких физических дисков в единый логический массив для повышения надежности;

RAID 5 — 5 уровень RAID;

JBOD — это технология объединения дисков без избыточности

CORS — это технология, которая с помощью специальных HTTP-заголовков позволяет веб-браузеру получать разрешение на доступ к ресурсам с сервера, находящегося на другом домене, отличном от текущего.

Фреймворк — это программная платформа, которая предоставляет базовую структуру для разработки приложений. Он включает в себя готовые компоненты, библиотеки, стандарты и инструменты, упрощающие создание программного обеспечения.

API — это набор строго определенных правил, протоколов и инструментов, которые позволяют различным программным компонентам взаимодействовать друг с другом.

ВВЕДЕНИЕ

Современные технологии хранения данных стремительно развиваются, предлагая пользователям всё более надёжные и эффективные решения для управления информацией. Одним из ключевых направлений в данной области является облачное хранение данных, которое обеспечивает доступ к файлам с любого устройства и высокую отказоустойчивость. Однако с ростом объёмов данных и требований к их безопасности возникает необходимость в оптимизации методов хранения, включая распределение информации между разными облачными сервисами для повышения надёжности.

Предлагаемое решение предоставляет пользователю доступ к файлам, расположенным на облачных сервисах, с возможностью восстановления данных при потере доступа к одному из сервисов.

В рамках работы предполагается разработка решения для взаимодействия с облачными сервисами, а также метода организации файловой системы по принципу RAID 5. Основой проекта выбрана технология ASP.NET Core, язык программирования C#. Для практической реализации и демонстрации работоспособности реализуется одностраничное веб-приложение.

Также в ходе работы требуется провести тестирование разработанного приложения. Необходимо оценить как корректность реализованного функционала, так и его соответствие выдвинутым к системе требованиям.

Целью данной бакалаврской работы является разработка приложения для хранения файлов на облачных сервисах с распределением по принципу RAID 5. Данное приложение должно представлять собой инструмент, позволяющий пользователю взаимодействовать с хранимыми данными.

Для достижения цели были поставлены следующие задачи:

1. Изучить принцип работы RAID 5;
2. Изучение различных сервисов хранения и способов взаимодействия с ними;
3. Реализовать модуль распределения данных по RAID 5;
4. Реализовать пользовательский интерфейс.

1 Описание предметной области

1.1 Актуальность поставленной задачи

В условиях стремительного роста объёмов цифровой информации, хранение данных в облаке становится всё более востребованным как в корпоративной, так и в частной сфере. Однако традиционные решения, предлагаемые облачными провайдерами, нередко страдают от недостаточной надёжности, высокой зависимости от одного поставщика и уязвимостей, связанных с компрометацией централизованных хранилищ.

Актуальной проблемой остаётся обеспечение отказоустойчивости и защиты пользовательских данных в облачных инфраструктурах, особенно при использовании бесплатных или условно-бесплатных сервисов с ограниченными гарантиями надёжности. Современные угрозы кибербезопасности и сбои в работе отдельных сервисов требуют разработки решений, которые не зависят от конкретного провайдера и обеспечивают сохранность информации даже в условиях частичной недоступности хранилищ.

Одним из подходов к решению данной задачи является использование принципов RAID-массивов на программном уровне, в частности уровня RAID 5, адаптированного под работу с облачными хранилищами. Распределённое хранение данных между независимыми сервисами, с возможностью их восстановления по контрольным суммам, позволяет достичь высокого уровня отказоустойчивости без необходимости в едином физическом хранилище.

Таким образом, разработка системы, реализующей RAID 5 с использованием облачных сервисов, отвечает актуальным требованиям к надёжности, доступности и безопасности данных. Подобный подход открывает перспективы создания дешёвых и масштабируемых решений для хранения критически важной информации без зависимости от конкретного поставщика.

У данной реализации можно выделить следующие преимущества:

1. В работе предложена реализация RAID 5 на уровне прикладной логики с использованием облачных хранилищ, предоставляемых независимыми провайдерами. Такой подход позволяет обеспечить отказоустойчивость при недоступности одного из сервисов и эффективно использовать ресурсы облака без необходимости физического RAID-контроллера.
2. За счёт распределения фрагментов файлов между различными провайдерами и хранения контрольной суммы обеспечивается повышенная

устойчивость к компрометации данных. Владелец части данных не может восстановить оригинальный файл, что повышает информационную безопасность.

3. Работа предлагает трёхуровневую архитектуру программной реализации RAID 5, включающую пользовательский интерфейс, бизнес-логику и уровень хранения, взаимодействующий с API облачных сервисов. Такой подход демонстрирует возможность программной эмуляции RAID в условиях облачной инфраструктуры.

1.2 Исследование инструментальных средств разработки

Для реализации серверной части приложения выбран язык программирования C# и кроссплатформенный фреймворк ASP.NET Core. Данный выбор обоснован следующими преимуществами:

- Язык C# является компилируемым и обладает оптимальной скоростью выполнения, что особенно важно при обработке данных и выполнении операций побитового XOR, необходимых для вычисления чётности в RAID 5.
- Встроенные средства языка и фреймворка позволяют эффективно обрабатывать сетевые запросы к API облачных хранилищ, обеспечивая отзывчивость и масштабируемость системы.
- ASP.NET Core обладает обширным набором инструментов для построения сервисов, соблюдающих стандарты, связанных с веб-запросами.
- Встроенные возможности ASP.NET Core по подключению дополнительных инструментов для тестирования и настройке прав доступа для запросов от внешних сервисов позволяют реализовать кросс-доменные взаимодействия с пользовательским интерфейсом и отладку API.

Для реализации пользовательского интерфейса был выбран фреймворк Vue 3. Его применение обусловлено следующими факторами:

- Данный фреймворк позволяет реализовать интерактивный интерфейс, не требующий полной перезагрузки страницы, что повышает удобство работы пользователя и снижает сетевые издержки.
- Язык программирования JavaScript, на котором основан данный фреймворк предоставляет инструментарий для взаимодействия с backend частью сервисов через HTTP-запросы, используя асинхронные методы.
- Выбранный инструмент предоставляет методы для тестирования и от-

ладки кода.

1.3 Обзор существующих решений

Современные технологии предлагают множество решений для облачного хранения, однако большинство из них имеет различные ограничения. Ниже приведён анализ существующих подходов и их слабых сторон:

1. Облачные хранилища с резервированием — Такие сервисы, как Amazon S3, Google Drive и Dropbox, предлагают встроенные механизмы резервирования данных внутри своей инфраструктуры. Однако они не обеспечивают распределение между независимыми провайдерами, что создаёт риск потери данных при сбое конкретного сервиса.
2. Программные эмуляции RAID — Существуют различные проекты, такие как StableBit CloudDrive, которые эмулируют RAID-массивы для облачных хранилищ. Однако они ограничены использованием одного типа сервиса и не поддерживают кросс-платформенное распределение, что является слабой стороной в вопросе конфиденциальности данных.

Конкретным примером актуальности разработки решения хранения данных на различных облачных сервисах является событие 29 ноября 2024 года. В этот день произошёл массовый сбой в работе сервисов Yandex Cloud. Данное событие повлекло за собой ряд сбоев в различных сервисах. Некоторые из них были недоступны, некоторым пришлось искать альтернативные решения. Данная ситуация подтверждает актуальность разработки решений для децентрализованного хранения данных у различных провайдеров.

Разрабатываемое приложение устраняет перечисленные недостатки за счёт:

- Распределения данных между разными облачными сервисами (Яндекс Диск, Dropbox, Google Drive), что снижает риски потери информации.
- Использования принципа RAID 5 для обеспечения отказоустойчивости без необходимости физических RAID-контроллеров.
- Решение предоставляет повышенную конфиденциальность при хранении данных, в случае если злоумышленник получит доступ к одному из хранилищ данных он не получит полную хранимую информацию.

2 Постановка задачи

Необходимо разработать одностраничное веб-приложение с распределением данных по принципу RAID 5 на трёх облачных сервисах для скачивания и загрузки файлов, а также восстановления данных в случае отказа одного из сервисов. Для реализации backend-части приложения используется фреймворк ASP.NET Core, язык программирования C#. Front-end часть приложения реализуется при помощи языка программирования JavaScript и фреймворка Vue 3. Для хранения данных используются сервисы Яндекс Диск, Dropbox, Google Drive.

Для реализации приложения были выделены следующие этапы:

- Реализация класса, отвечающего за сборку проекта, вносящего необходимые при сборке проекта параметры.
- Разработка класса, выполняющего инициализацию представленных в проекте сервисов
- Реализация классов, отвечающих за запросы на облачные сервисы
- Подготовка набора сервисов, отвечающих за внутреннюю логику, выполняющих запросы на пользовательский интерфейс
- Реализация контроллеров для взаимодействия с внешними приложениями

Основной функционал программы подразумевает:

- Загрузка данных — Позволяет загрузить файл в облачные сервисы с учетом распределения данных.
- Скачивание файла — Позволяет скачать разделенный файл с автоматической сборкой файла.
- Восстановление данных — Позволяет восстановить утерянные данные в случае отказа одного из дисков.
- Сборка файла из частей — При распределении при помощи RAID 5 файл хранится в виде нескольких наборов байтов.
- Разделение массива данных файла на части — Для распределения при помощи RAID 5 файл должен быть разделен на несколько наборов байтов.
- Применение массива чётности — Для восстановления утерянных данных используется массив чётности.
- Запрос к облачным сервисам — Воспроизведение взаимодействия при-

ложения с облачными сервисами.

Схема взаимодействия пользователя с программой предоставлена на рисунке 1.

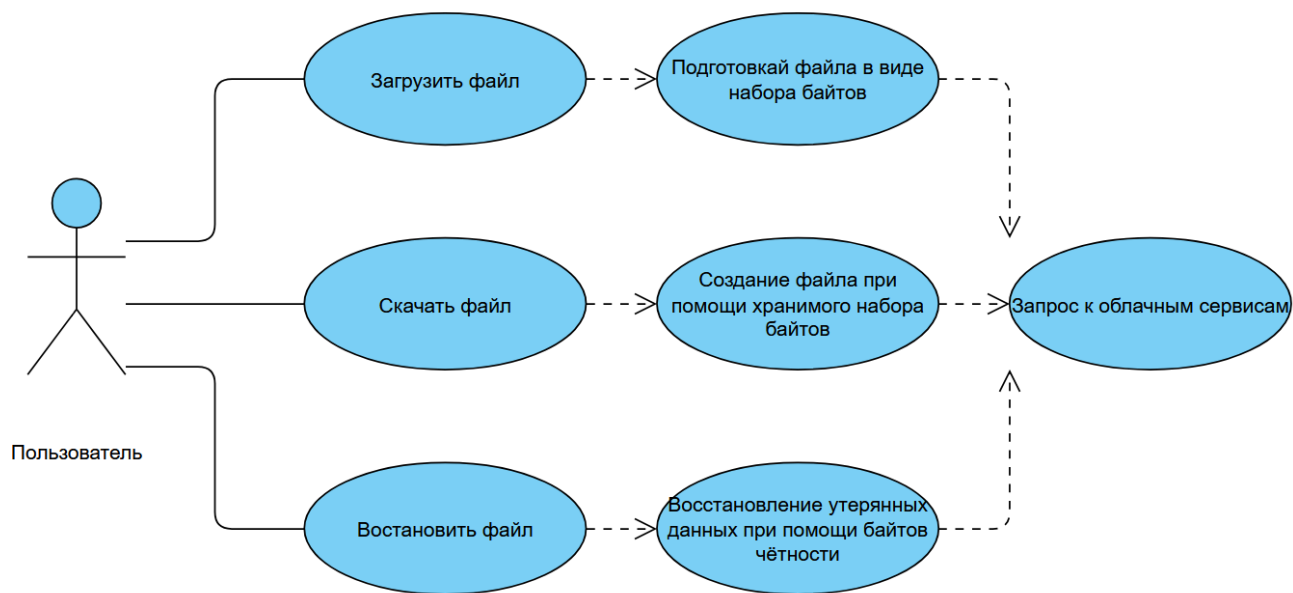


Рисунок 1 – Схема взаимодействия пользователя с программой

3 Теоретические сведения

3.1 Общие сведения о RAID

Технология RAID появилась в эпоху, когда жесткие диски были значительно дороже и менее надежными, чем сейчас. Изначально аббревиатура RAID имела расшифровку "Избыточный массив недорогих дисков" (Redundant Array of Inexpensive Disks), но со временем ее значение изменилось на "Избыточный массив независимых дисков" (Redundant Array of Independent Disks). Системы, поддерживающие RAID, часто называют RAID-массивами [1].

Основные задачи RAID:

1. Повышение производительности за счет чередования (striping) – данные распределяются по нескольким дискам, что снижает нагрузку на каждый из них.
2. Увеличение отказоустойчивости благодаря избыточности (redundancy) – даже при отказе одного диска система продолжает работу за счет резервных данных.

Отдельный жесткий диск имеет ограниченную скорость и срок службы, но объединение нескольких дисков в RAID-массив позволяет значительно улучшить надежность и производительность системы в целом.

RAID-контроллер объединяет физические диски в виртуальный жесткий диск, который сервер воспринимает как единое устройство. При этом реальное распределение данных между дисками остается скрытым и видно только администратору.

Существуют разные уровни RAID, определяющие способы распределения данных. Почти все они предусматривают избыточное хранение информации, что позволяет восстановить данные при отказе диска. Восстановление данных происходит параллельно с работой сервера, что может временно снизить производительность [2].

RAID-массивы различаются по способу организации данных и уровню избыточности. Каждый уровень RAID определяет:

- метод записи (чередование, зеркалирование, контрольные суммы).
- степень отказоустойчивости.
- производительность.
- эффективность использования дискового пространства.

Далее будут рассмотрены наиболее распространённые в применении

уровни RAID

RAID 0 (Striping – чередование)

- Принцип работы: Данные разбиваются на блоки и равномерно распределяются по всем дискам массива.
- Преимущества:
 - Максимальная производительность (чтение/запись ускоряются за счет параллельной работы дисков).
 - Полное использование емкости (нет избыточности).
- Недостатки:
 - Нет отказоустойчивости — выход одного диска приводит к потере всех данных.
- Минимальное количество дисков: 2
- Применение: Для временных данных, кэширования, задач, где важна скорость, но не надежность.

RAID 1 (Mirroring – зеркалирование)

- Принцип работы: Данные полностью дублируются на двух или более дисках.
- Преимущества:
 - Высокая надежность — при отказе одного диска данные сохраняются на другом.
 - Быстрое восстановление.
- Недостатки:
 - Высокие затраты — полезная емкость равна половине общего объема (при двух дисках).
 - Скорость записи может уменьшаться за счет количества дисков.
- Минимальное количество дисков: 2
- Применение: Критически важные данные.

RAID 5 (Чередование с контролем чётности)

- Принцип работы: Данные и контрольные суммы (parity) распределяются по всем дискам.
- Преимущества:
 - Оптимальный баланс между надежностью, производительностью и затратами.
 - Выдерживает отказ одного диска без потери данных.

- Эффективное использование емкости.
- Недостатки:
 - Замедление записи из-за расчета контрольных сумм.
 - Долгое восстановление при замене диска
- Минимальное количество дисков: 3
- Применение: Файловые серверы, веб-хранилища, СУБД RAID 6 (Двойная четность)
- Принцип работы: Данные и два набора контрольных сумм распределяются по всем дискам.
- Преимущества:
 - Выдерживает отказ двух дисков одновременно.
 - Надежнее RAID 5 для больших массивов.
- Недостатки:
 - Еще более низкая скорость записи
 - Большие потери емкости
- Минимальное количество дисков: 4
- Применение: Системы с высокими требованиями к отказоустойчивости RAID 10 (1+0, зеркалирование + чередование)
- Принцип работы: Комбинация RAID 1 и RAID 0 — сначала диски зеркалируются, затем данные чередуются.
- Преимущества:
 - Высокая производительность
 - Хорошая отказоустойчивость
- Недостатки:
 - Высокая стоимость (только половина полезной емкости).
- Минимальное количество дисков: 4
- Применение: Высоконагруженные базы данных, виртуализация, транзакционные системы.

В таблице 1 предоставлена сравнительная характеристика для рассмотренных уровней RAID. Эффективность занимаемого места отображает то, сколько места от изначального объема хранилища данных займет организация RAID-массива. В реализации моделей RAID 5 и RAID 6 объем данных, резервируемый под организацию распределения, составляет соответственно 1 и 2 размера хранилища, в следствии чего N при вычислении эффективно-

сти — это количество использованных дисков.

Таблица 1 – Сравнительная характеристика уровней RAID

Уровень RAID	Эффективность занимаемого места	Скорость чтения	Скорость записи	Отказоустойчивость
RAID 0	100%	Высокая	Высокая	Нет
RAID 1	50%	Средняя	Низкая	Высокая
RAID 5	$(N-1)/N$	Высокая	Средняя	Средняя
RAID 6	$(N-2)/N$	Высокая	Низкая	Очень высокая
RAID 10	50%	Очень высокая	Высокая	Очень высокая

Исходя из предоставленных данных, можно сделать вывод об эффективности методов для реализации поставленной задачи. RAID 5 представляется наиболее сбалансированной моделью для применения на небольших данных, является экономной по занимаемому месту, резервируя только одно из предоставленных хранилищ данных, предоставляет исчерпывающую отказоустойчивость, позволяя восстановить данные при потере доступа к одному из дисков, а также в предложенной реализации обеспечивает конфиденциальность данных за счет распределения информации на нескольких сервисах. Последнее преимущество является особенно примечательным, поскольку в случае взлома одного из хранилищ злоумышленник не сможет получить первоначальные данные пользователя.

3.2 RAID 5

RAID 5 уровня является отказоустойчивым массивом хранения данных на нескольких хранилищах данных, в данном случае на облачных сервисах, представляющим собой распределение данных на определенное количество чередований с вычислением для них контрольных сумм. Контрольная сумма представляет собой $A \oplus B = Parity$, где A и B представляют собой два массива байтов разделённого файла [3]. Таким образом, достигается возможность восстановить данные при потере A или B ввиду следующих равенств:

$$— A \oplus Parity = B$$

$$— Parity \oplus B = A$$

Как правило, файл разделяется по одному из двух правил, в зависимости от размера данных, с которыми предстоит работать:

1. Весь файл целиком делится на 6 равных по размеру массивов байтов,

данные массивы разделяются на 3 пары и для каждой пары вычисляется чётность.

2. Задаётся максимальный размер массива и данные распределяются до тех пор, пока весь файл не будет разделен и их количество не будет кратно 6.

Например, имеется файл, который требуется распределить вторым методом, для этого файл поочередно разделяется n раз на части в формате массивов байт $A_i, B_i, C_i, D_i, E_i, F_i$. Из данных массивов формируются пары $A_i B_i, C_i D_i, E_i F_i$, для каждой из них формируются массивы чётности:

- $A_i \oplus B_i = Parity_{i1}$
- $C_i \oplus D_i = Parity_{i2}$
- $E_i \oplus F_i = Parity_{i3}$

Затем данные распределяются между дисками так, чтобы на диске сохранилась ровно одна чётность данного чередования, а на двух других соответствующие данные пары. Процесс повторяется до тех пор, пока файл не будет исчерпан.

На рисунке 2 продемонстрирована модель распределения данных с применением трёх дисков.

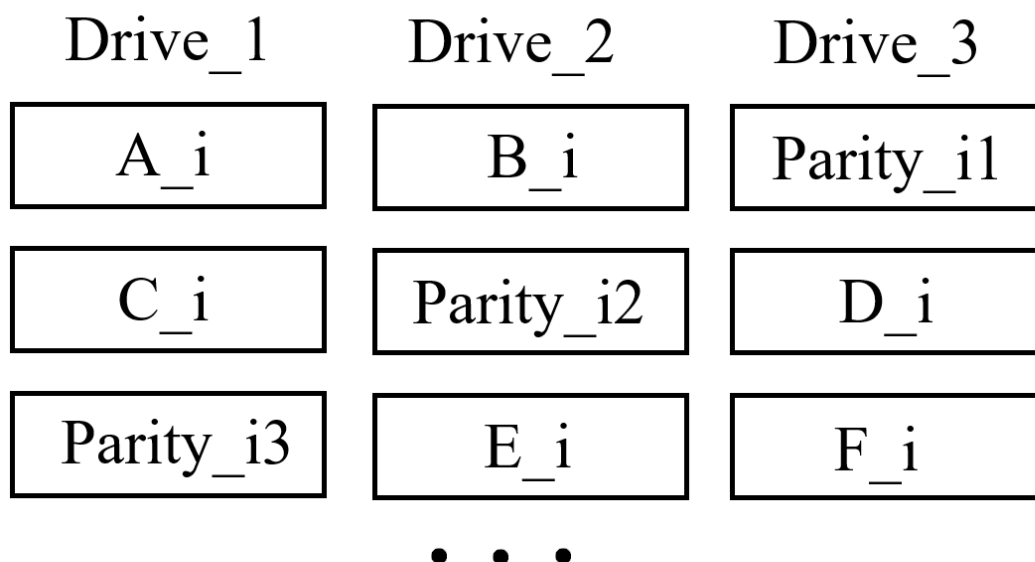


Рисунок 2 – Модель распределения данных

Таким образом, при выводе из строя одного из дисков, например, диска

2, все утерянные данные можно восстановить до первоначальных массивов с байтами следующим образом:

$$— A_i \oplus Parity_{i1} = B_i$$

$$— Parity_{i3} \oplus F_i = E_i$$

Данный метод предполагает, что будет использоваться как минимум 3 диска, при этом имея возможность масштабируемости на необходимое большое количество. Потери в общем объеме хранилища будут составлять объем одного диска, при этом данные подлежат восстановлению при потере одного хранилища.

4 Разработка решения для обработки данных на облачных сервисах

4.1 Использованные технологии

Для разработки серверной части проекта был выбран разработанный компанией Microsoft фреймворк `ASP.Net Core`. Данный фреймворк представляет технологию для создания веб-приложений на платформе `.NET`. Предоставленная технология обеспечивает средства для разработки серверной части веб-приложения. Также он предоставляет инструментарий для подключения дополнительных библиотек, обеспечивает подключение для взаимодействия с веб-интерфейсами. В качестве языка программирования для разработки приложений при помощи `ASP.NET Core` применяется компилируемый язык программирования `C#`.

При распределении файлов по принципу `RAID 5` требуется вычисление контрольных сумм для обеспечения отказоустойчивости. Это интенсивная операция, зависящая от вычислений на процессоре, где компилируемые языки имеют значительное преимущество перед интерпретируемыми. При вычислении операции $A \oplus B$ компилируемый язык преобразует её в одну инструкцию для процессора, в свою очередь, интерпретируемый сначала произведёт проверку типов данных переменных, затем определит перегруженный оператор, отвечающий за данную операцию, и в конце начнёт выполнение соответствующего метода.

При разработке приложения был применён принцип трёхуровневой архитектуры. Это классический подход к проектированию программных систем, в котором приложение делится на три логических слоя:

1. Уровень представления — Этот слой отвечает за взаимодействие с пользователем и отображение данных. Он обеспечивает коммуникацию между пользователем и системой. В данном случае за него отвечает пользовательский интерфейс. Он выполняет следующие функции:
 - Загрузка файла на облачные сервисы с распределением по принципу `RAID 5`
 - Скачивание файла с облачных сервисов
 - Возможность восстановить потерянные данные, в случае вывода из строя одного из дисков
2. Уровень логики приложения — Этот слой отвечает за бизнес-логику и

правила работы системы. Он обрабатывает запросы, поступающие с уровня представления, и выполняет всю логику, необходимую для обработки данных, взаимодействия с базами данных и других операций. Он реализует следующие функции:

- Распределение данных, хранящихся в выбранном файле в виде массива байтов, готового для отправления.
- Подготовка массива чётности, реализующего восстановление данных, в случае отказа в доступе к одному из облачных хранилищ.
- Сборка массивов байтов, распределённых по принципу RAID 5 в виде файла

3. Слой данных — Этот слой отвечает за управление данными и их хранение. Он взаимодействует с базами данных, файлами, облачными хранилищами или внешними сервисами данных. Слой данных получает запросы от бизнес-логики для чтения или изменения данных. Данный слой реализует взаимодействие с представленными облачными хранилищами:

- Яндекс Диск
- Dropbox
- Google Drive

Данный подход к проектированию предоставляет преимущества для реализации распределённого облачного хранилища. Чёткое разделение на уровни позволяет независимо модифицировать интерфейс пользователя, бизнес-логику обработки файлов и механизмы взаимодействия с облачными сервисами. Кроме того, трёхуровневая структура упрощает тестирование каждого компонента системы по отдельности и способствует поддержанию чистоты кода.

Основной класс, отвечающий за сборку проекта, представляет собой точку входа **ASP.NET Core** приложения и содержит конфигурацию веб-сервера. Установка конфигурации производится путём передачи параметров в экземпляр класса, отвечающего за инициализацию. При сборке проекта были добавлены два инструмента:

- Swagger
- CORS

Swagger представляет собой набор инструментов для тестирования и

визуализации запросов [4]. Основная часть данного инструмента при работе с API — предоставляет возможности интерактивного тестирования разработанных сервисов. Он обеспечивает доступ к проведению запросов напрямую из браузера, без использования вручную разработанного интерфейса, показывает разработчику все необходимые поля с типами данных.

Также данный инструмент предоставляет возможность создания описаний работы API: информации о ресурсах, параметрах запросов, возвращаемых данных и сведениях о результатах запросов. Чтобы автоматизировать это описание, сделать его структурированным и прозрачным.

Следующим подключенным параметром является Cross-Origin Resource Sharing (CORS) — это технология, которая с помощью специальных HTTP-заголовков позволяет веб-браузеру получать разрешение на доступ к ресурсам с сервера, находящегося на другом домене, отличном от текущего. Если веб-страница пытается запросить данные с другого источника (cross-origin HTTP-запрос), это означает, что домен, протокол или порт запрашиваемого ресурса отличаются от тех, что указаны в исходном документе [5].

Например, если HTML-страница с сервера `http://domain-a.com` запрашивает изображение ``, это будет считаться кросс-доменным запросом. Многие сайты загружают ресурсы (например, CSS, изображения, скрипты) с различных доменов, особенно при использовании CDN (Content Delivery Networks).

В целях безопасности браузеры накладывают ограничения на кросс-доменные запросы, выполняемые через JavaScript, следуя политике единого источника. Это означает, что веб-приложение может запрашивать ресурсы только с того домена, с которого оно было загружено, если сервер явно не разрешит доступ через CORS.

CORS обеспечивает безопасный обмен данными между браузером и серверами при кросс-доменных запросах. Современные браузеры используют этот механизм в API (таких как XMLHttpRequest и Fetch), чтобы минимизировать риски, связанные с межсайтовыми запросами.

ASP.NET Core предоставляет возможность настройки разрешений для данной технологии. Для настройки политики доступа необходимо передать в метод адрес, с которого будут приходить запросы, а также указать название для текущей конфигурации [6]. Пример использования продемонстрирован

на листинге ниже:

```
builder.Services.AddCors(options =>
{
    options.AddPolicy("AllowVueApp",
        policy => policy.WithOrigins("http://localhost:5173")
            .AllowAnyHeader()
            .AllowAnyMethod());
});
```

4.2 Яндекс Диск

Яндекс Диск является одним из выбранных облачных хранилищ для разработки. Он предоставляет набор инструментов для разработчиков, позволяющий получать данные пользователя с его согласия, а также взаимодействовать с его личным хранилищем. Для применения данного диска в разработке RAID можно выделить следующие этапы:

1. Создание приложения в сервисе Яндекс ID;
2. Получение токена пользователя;
3. Разработка методов для взаимодействия с хранилищем пользователя.

При создании приложения необходимо указать его название, логотип и выбрать тип устройств, на которых оно будет предположительно исполняться. Далее следует указать адрес, на который пользователь будет перенаправлен для авторизации. Выбор данного адреса зависит от метода получения токена, для применения в веб-сервисе, как правило, используются технологии мгновенной авторизации. Для этого адресом используется ссылка, по которой создаётся вспомогательная страница для приёма токена [7].

Стандартное приложение, созданное в Яндекс ID предоставляет возможность запрашивать следующие данные аккаунта при регистрации:

- Логин, имя и фамилия, пол.
- Портрет пользователя.
- Адрес электронной почты.
- Номер телефона.
- Дата рождения.

Однако, данной информации недостаточно для достижения поставленной цели, в следствие чего к сервису необходимо подключить REST API, зна-

чительно расширяющий набор предоставляемых данных. Для реализации данного приложения были выбраны пункты чтения всего диска и записи в любое место диска.

В завершение работы с данным сервисом необходимо получить **OAuth** токен пользователя, необходимый в **http** запросах с сервера. Существует несколько методов, однако в данной работе будет рассмотрен метод получения отладочного токена [8]. Для этого необходимо выполнить следующий ряд действий:

1. Пользователь должен предоставить приложению доступ по указанному адресу перенаправления.
2. Изменить указанный адрес, добавив в конце `client_id=<идентификатор приложения>`. Данное значение можно узнать на панели управления сервисом.
3. Затем откроется страница с подтверждением доступа и будет выдан текущий токен.

Для отправления запросов использовался класс **HttpClient** — это класс в **.NET**, предназначенный для отправки **HTTP**-запросов и получения **HTTP**-ответов от ресурсов, идентифицируемых **URI**. Основные его функции:

- Отправка **HTTP**-запросов (**GET**, **POST**, **PUT**, **DELETE**)
- Работа с заголовками запросов и ответов
- Управление временем ожидания и политиками повторов
- Поддержка асинхронных операций
- Возможность обработки различных форматов данных (**JSON**, **XML** и др.)

Выгрузка файла осуществляется при помощи запроса с переданными полями **OAUTH** токен, путь к файлу на облачном сервисе и массив байт файла для отправки. Изначально запрашивается ссылка для загрузки файла. Полученный ответ хранится в **JSON** формате, из-за чего его требуется десериализация. Далее формируется массив байтов с файлом и вместе с полученной ранее ссылкой формируется запрос для выгрузки. Пример данного кода приведён в листинге ниже:

```
public async Task<string> UploadFile
    (string filePath, string localFilePath)
{
```

```

var uploadUrlResponse = await _httpClient
    .GetAsync($"{_apiUrl}resources/upload?path={filePath}
        &overwrite=true&fields=name,_embedded.items.path");
    // URL для загрузки файла
uploadUrlResponse.EnsureSuccessStatusCode(); // Статус запроса

// Десериализация ответа с адресом для загрузки
var uploadUrl = JsonConvert
    .DeserializeObject<YandexDiskUploadResponse>(await
        uploadUrlResponse.Content.ReadAsStringAsyncAsync()).Href;

byte[] fileContent = File.ReadAllBytes(localFilePath);

var content = new ByteArrayContent(fileContent);
// Загрузка файла по полученному URL
var uploadResponse = await _httpClient
    .PutAsync(uploadUrl, content);
uploadResponse.EnsureSuccessStatusCode();
// Проверка, что загрузка прошла успешно

return await uploadResponse.Content.ReadAsStringAsyncAsync();
// Результат загрузки
}

```

Следующей подзадачей является скачивание файла с облачного сервиса. В виде параметров для запроса передаются OAuth токен, путь к файлу, расположенному в хранилище. В данном случае файл будет храниться в корневой папке хранилища, в связи с чем путём будет являться название файла. Формируется запрос с путём к файлу на облачном сервисе. Полученная ссылка десериализуется из формата JSON и применяется в запросе для загрузки файла. Полученный файл конвертируется из массива байтов в изначальный вид. Пример продемонстрирован на листинге ниже:

```

public async Task<byte[]> DownloadFile(string filePath)
{

```

```

var downloadUrlResponse = await _httpClient
    .GetAsync($"{_apiUrl}resources/download?path={filePath}");
    // URL для загрузки файла
downloadUrlResponse.EnsureSuccessStatusCode();
// Статус запроса
var downloadUrl = JsonConvert
    .DeserializeObject<YandexDiskDownloadResponse>
    (await downloadUrlResponse.Content
        .ReadAsStringAsync()).Href;
// Десериализация ответа с адресом для скачивания
var downloadResponse = await _httpClient
    .GetAsync(downloadUrl);
// Скачивание файла по полученному URL
downloadResponse.EnsureSuccessStatusCode();
// Проверка результата запроса

return await downloadResponse.Content
    .ReadAsByteArrayAsync(); // Содержимое файла
}

```

4.3 Dropbox

Dropbox является облачным сервисом, позволяющим получать данные пользователя и взаимодействовать с его хранилищем данных [9]. Для применения данного сервиса были поставлены следующие цели:

1. Регистрация сервиса для облака;
2. Получение токена пользователя;
3. Подготовка класса для взаимодействия с диском пользователя.

Для создания приложения необходимо перейти в консоль разработчика. Выбрав соответствующую опцию, будут запрошены следующие параметры:

- Тип API
- Уровень доступа к диску
- Название регистрируемого в консоли приложения

При выборе уровня доступа к пользовательским данным присутствуют две опции. Папка для приложения — для приложения будет использоваться

конкретное место хранения данных в облачном хранилище. Полный доступ — сервис будет иметь доступ ко всей информации, хранящейся на диске. После регистрации сервиса нужно указать, какие конкретно права будут предоставлены. Для данной реализации были выбраны следующие параметры:

- `files.metadata.write` — Редактирование файлов и папок на диске;
- `files.metadata.read` — Просмотр файлов и папок на диске;
- `files.content.write` — Редактирование данных, связанных с файлами на диске;
- `files.content.read` — Просмотр данных, связанных с файлами на диске;

Генерация токена доступа может осуществляться двумя способами. Он может быть создан вручную в меню приложения, данный токен будет действителен в течение двух недель. Также возможна автоматическая генерация при помощи запроса на сервере. В данном случае пользователю необходимо подтвердить предоставление прав на использование данных приложением, перейдя на специальный адрес [10].

Для формирования запроса необходимо иметь токен доступа, а также уникальный адрес, идентифицирующий текущий запрос. В данной работе будут применены два запроса: скачивание файла и загрузка на диск. Запрос для скачивания файла принимает только его путь на облачном сервисе, загрузка файла имеет следующие параметры:

- `path` — Путь в Dropbox пользователя для сохранения файла.
- `mode` — Определяет модель поведения, если файл уже существует. По умолчанию для этого параметра — добавить файл.
- `autorename` — Является булевым значением. Если возникает конфликт (согласно настройке параметра `mode`), сервер Dropbox попытается автоматически переименовать файл, чтобы избежать конфликт. По умолчанию - `False`.
- `client_modified` — Является необязательным полем, позволяет указать фактическое время создания файла, помимо автоматически записываемого значения времени загрузки.
- `mute` — Булево значение, отвечает за получение пользователем уведомлений об изменениях файлов. Если установлено значение `True`, то клиент не будет получать оповещение. По умолчанию значение `False`.
- `property_groups` — Необязательное поле, позволяет добавить файлу поль-

зовательские свойства.

- `strict_conflict` — Является булевым значением. Представляет собой более строгую проверку каждой записи вызывает конфликт, даже если целевой путь указывает на файл с идентичным содержимым.
- `content_hash` — Не обязательный параметр. Хеш содержимого загружаемого файла. Если указан и загруженное содержимое не соответствует этому хешу, будет возвращена ошибка.

Реализация серверной части представляет собой конструктор и набор методов, совершающих запросы для скачивания и загрузки файлов. Конструктор класса представляет собой инициализацию экземпляра `httpClient`, токена доступа и шаблона адреса.

Метод загрузки файла принимает адрес файла, хранящегося на устройстве, а также путь к файлу на удалённом хранилище. В данной реализации все файлы будут храниться в корневой папке, в следствие чего путь к файлу будет обозначаться как строка `"/` объединённая с названием файла. Файл представляется в виде массива байтов. При формировании запроса используется метод `POST` с используемым для загрузки адресом. Также в запросе используются следующие поля:

- `path`
- `mode`
- `autorename`
- `mute`

Затем к запросу добавляется содержимое файла. Выполняется его выполнение и проверка статуса. На сервер возвращается ответ от сервиса. Пример данной реализации предоставлен на листинге ниже:

```
public async Task<string> UploadFile
(string localFilePath, string dropboxPath){
    byte[] fileContent = File.ReadAllBytes(localFilePath);
    \\Массив байтов файла
    var request = new HttpRequestMessage \\Тело запроса, POST,
    (HttpMethod.Post, $"{_apiUrl}upload"); \\тип загрузка файла
    request.Headers.Add("Dropbox-API-Arg", \\Параметры запроса
        JsonConvert.SerializeObject(new
        {
```



```

        path = dropboxPath, \\Путь к файлу
        mode = "overwrite", \\Перезапись если файл существует
        autorename = true, \\Разрешение конфликта названий
        mute = false \\Сохранение уведомлений
    }));
request.Content = content; \\Передача файла в тело запроса
var response = await _httpClient.SendAsync(request);
response.EnsureSuccessStatusCode(); \\Исполнение запроса
return await response.Content.ReadAsStringAsync();
\\Ответ сервиса
}

```

Для скачивания файла передаётся путь, по которому он хранится на Dropbox. Формируется запрос типа GET с адресом для загрузки файла, затем формируются параметры, в которые указывается путь к файлу. После этого выполняется запрос, полученные данные десериализуются и преобразовываются в виде файла из массива байт. Пример данной реализации предоставлен на листинге ниже:

```

public async Task<byte[]> DownloadFile(string dropboxPath)
{
    var request = new HttpRequestMessage \\Формирование запроса
        (HttpMethod.Post, $"({_apiUrl}download)");
    request.Headers.Add("Dropbox-API-Arg",
        JsonConvert.SerializeObject(new \\Настройка параметров
        {
            path = dropboxPath \\Путь к файлу
        }));
    var response = await _httpClient.SendAsync(request);
    \\Выполнение запроса
    response.EnsureSuccessStatusCode(); \\Статус запроса
    var result =
        JsonConvert.DeserializeObject<DropboxFileMetadata>(
            response.Headers.GetValues("Dropbox-API-Result").First());
    \\Десериализация данных в массив байт
}

```

```
return await response.Content.ReadAsByteArrayAsync();  
}
```

4.4 GoogleDrive

Третьим выбранным облачным хранилищем является Google Drive. Данный сервис предоставляет полноценную библиотеку для .NET, реализующую взаимодействие с диском при помощи зарегистрированного приложения.

Установка данного пакета в среде Visual Studio производилась при помощи NuGet Package Manager. Данный инструмент позволяет устанавливать необходимые для проекта библиотеки, отслеживать изменения в версиях, а также производить изменения в установленных пакетах, отключать и переносить их [11]. Для применения были установлены следующий набор библиотек:

- Google.Apis — API для взаимодействия с сервисами Google, формирования запросов.
- Google.Apis.Drive.v3 — Является библиотекой для работы с диском.

Для взаимодействия с диском пользователя необходимо зарегистрировать клиент в сервисе Google Auth Platform. При создании приложения необходимо указать его название и платформу, для которой ведётся разработка. Среди них предоставляется выбор:

- Веб-приложение
- Приложение на систему Android
- Расширение для браузера
- Приложение на систему IOS
- Приложение для телевизоров и устройств с ограниченной гарнитурой
- Оконное приложение

Также необходимо указать адрес, на который будет перенаправлен пользователь для авторизации в приложении. Данный адрес должен совпадать с адресом, на котором работает сервер. Используемая библиотека при авторизации запроса запускает отдельный клиент, выбирая случайный свободный порт, в результате чего он должен оставаться пустым для автоматического определения.

Google Drive разделяет запрашиваемые данные на три уровня безопасности. В зависимости от максимального среди затронутых уровней на приложение могут накладываться ограничения. При авторизации пользователь будет уведомлен о том, насколько безопасно использовать данное приложе-

ние. В случае, если при разработке запрашиваются данные ограниченной области действия, пользователей можно добавлять только напрямую через консоль клиента до подтверждения безопасности приложения [12]. Под особо чувствительные запросы попадают:

- Просмотр и скачивание файлов
- Изменение содержимого данных
- Просмотр информации о файлах

Формирование запроса происходит при помощи файла с учетными данными пользователя. Данный файл может быть сформирован в меню приложения. Он хранит в себе следующие поля:

- `client_id` — Уникальный идентификатор приложения
- `project_id` — Идентификатор проекта в Google Cloud Platform, к которому привязаны учетные данные
- `auth_uri` — URL-адрес, куда направляется пользователь для авторизации
- `token_uri` — URL-адрес, на который направляется запрос для обмена `authorization code` на `access token` и `refresh token`
- `auth_provider_x509_cert_url` — Ссылка на сертификаты x509, используемые для проверки подлинности OAuth 2.0
- `client_secret` — Закрытый ключ
- `redirect_uris` — Список URI, на которые Google может перенаправлять пользователя после авторизации
- `javascript_origins` — Домены, с которых разрешены JavaScript-запросы к API Google

Класс, реализующий логику взаимодействия с облачным сервисом, имеет поля, отвечающие за путь к файлу с учётными данными пользователя, а также идентификатор папки, которая будет просматриваться для взаимодействия. Конструктор класса представляет собой инициализацию всех объявленных полей. Для применения данных пользователя в запросах применяется класс `UserCredential`, предоставленный библиотекой [13].

Метод загрузки файлов принимает путь к файлу на устройстве пользователя. При помощи него определяется название файла, его содержимое записывается в виде массива байтов. Используя полученные пользовательские данные, формируется экземпляр класса `GoogleClientSecrets`, при помощи

которого запускается клиент для авторизации пользователя и дальнейшего взаимодействия с диском [14]:

```
UserCredential credentials;
var clientSecrets = await
    GoogleClientSecrets.FromFileAsync(credentialsPath);

credentials = await
    GoogleWebAuthorizationBroker.AuthorizeAsync(
        clientSecrets.Secrets,
        new[] { DriveService.ScopeConstants.DriveFile },
        "user",
        CancellationToken.None);
```

Затем создается экземпляр сервиса для формирования запроса, он хранит в себе поля с названием созданного приложения, а также клиент для формирования запроса. Загрузка файла требует указания определенной информации о нём, а именно его название и папку, к которой он принадлежит, поэтому создается отдельная переменная, хранящая поля с данными:

```
var service = new DriveService(new BaseClientService.Initializer()
{
    HttpClientInitializer = credentials,
    ApplicationName = "Raid5"
});
var fileMetaData = new Google.Apis.Drive.v3.Data.File()
{
    Name = Path.GetFileName(localFilePath),
    Parents = new List<string> { folderId }
};
```

В итоге открывается поток для чтения файла, который передается в метод **Create**, ожидающий поля с данными о файле, его содержимом в виде потока и типа данных. Данный метод подготавливает запрос с файлом для отправки. Файл асинхронно загружается на диск и по завершении возвращает статус операции:

```

using (var stream = new FileStream(localFilePath, FileMode.Open))
{
    request = service.Files.Create(fileMetaData, stream, "");
    request.Fields = "id";
    var progress = await request.UploadAsync();
    if (progress.Status != UploadStatus.Completed)
        throw new Exception($"Upload failed:
        {progress.Exception?.Message}");
}

```

Для реализации загрузки файла с облачного диска необходимо получить его идентификатор. В данном случае формируется запрос, запрашивающий список всех файлов в данной папке на диске, после чего файлы фильтруются по указанному названию и проверяются, чтобы они не считались удалёнными. В итоге он исполняется и возвращает первое подходящее под условия вхождение:

```

var request = service.Files.List();
request.Q = $"name = '{fileName}' and trashed = false";
request.Fields = "files(id, name)";

var result = await request.ExecuteAsync();
return result.Files.FirstOrDefault()?.Id;

```

Скачивание файлов производится аналогичным образом. Производится подготовка клиента для формирования запроса, однако в этот раз для него необходимо получить идентификатор. Для этого применяется метод, подготовленный `FindFileIdByName`, он принимает на вход название и подготовленный экземпляр сервиса. Данный метод возвращает уникальный идентификатор файла, находя его по названию в выбранной папке. Затем открывается поток для сохранения файла на устройстве пользователя, в теле которого исполняется сам запрос. Пример приведён на листинге ниже:

```

var fileId = await FindFileIdByName(fileName, service);
var fileInfo = await service.Files.Get(fileId).ExecuteAsync();
if (fileInfo == null)

```

```

        throw new Exception("File not found in Google Drive");
using (var fileStream = new
FileStream(localSavePath, FileMode.Create, FileAccess.Write))
{
    var request = service.Files.Get(fileId);
    await request.DownloadAsync(fileStream);
}

```

В данном классе подготовлен метод для получения названий всех файлов, который необходим для корректного отображения информации в пользовательском интерфейсе. При выполнении запроса собирается список из названий всех файлов, хранимых в указанной папке. Пример приведён на листинге ниже:

```

var request = service.Files.List();
request.Q = $"''{folderId}' in parents and trashed = false";
request.Fields = "files(name)";

var result = await request.ExecuteAsync();
return result.Files.Select(file => file.Name).ToList();

```

5 Реализация класса, распределяющего данные среди облачных хранилищ по принципу RAID 5

Данная реализация представляет собой класс, содержащий в себе набор методов для взаимодействия со всеми предоставленными облачными хранилищами. Список всех реализованных методов:

- Загрузка данных с распределением на все диски
- Скачивание файла, при условии что все диски доступны
- Скачивание данных, если не доступен Яндекс Диск
- Скачивание данных, если не доступен Dropbox
- Скачивание данных, если не доступен Google Drive
- Расчет чётности для двух массивов байт

RAID 5 уровня подразумевает распределение файлов среди трёх и более дисков с подготовкой чётности для их элементов. В данном решении файл будет представляться в виде нескольких массивов байт для организации RAID, а каждый массив будет преобразован в файл для хранения данных на хранилищах. Операция преобразования в файл добавляет дополнительные вычисления, ввиду чего был применён способ, в котором файл представляется в виде массивов байтов и делится целиком на равные части а также формируются пары с массивами чётности для каждой. Данный подход минимизирует повторение операций при работе с облачными хранилищами.

Так как для организации RAID 5 необходимо представить файл в виде шести равных по объему частей возникает проблема с ситуацией, когда количество байт в файле не является кратным. Для решения данной проблемы необходимо увеличить размер каждой части так, что бы их суммарный объем был кратен шести, а так же наиболее приближен к изначальному размеру файла в целях экономии места на хранилище.

Данный вопрос был решен следующим образом: при расчете размера каждой части производится деление на 6 с округлением вверх. Таким образом их сумма будет ближайшим числом кратным шести, а также она будет больше, либо равно изначальному числу. Однако при таком подходе возникает следующая проблема: байтов в частях стало больше чем в изначальном файле.

Для её решения был подготовлен следующий метод: рассчитывается отдельное число, являющееся количеством лишних байт в файле. Так как

производится деление с округлением вверх, для его получения необходимо вычесть из шести остаток от деления изначального размера на шесть. Полученное число отображает количество лишних байт в файле и оно так же будет храниться на диске в виде файла размером в пару байт для восстановления файла в его изначальный вид.

По скольку в ситуации, когда размер файла не кратен шести, размер изначального файла расширяется, в последнем фрагменте данных не будет хватать байтов для заполнения. В данном случае недостающие байты заполняются нулями. Эта операция производится для возможности восстановления данных в случае недоступности одного из хранилищ так как эти данные не повлияют на результат операции XOR. При сборке файлов лишние нулевые байты не используются.

На рисунке 3 продемонстрирован пример конкретной реализации распределения данных, представленной в программе. $A_1, A_2, B_1, B_2, C_1, C_2$ — наборы байт изначального файла, A_P, B_P, C_P — массивы четности, A_R, B_R, C_R — байты, хранящие информацию о лишних байтах. Все приведенные массивы хранятся на облачных сервисах в виде файлов.

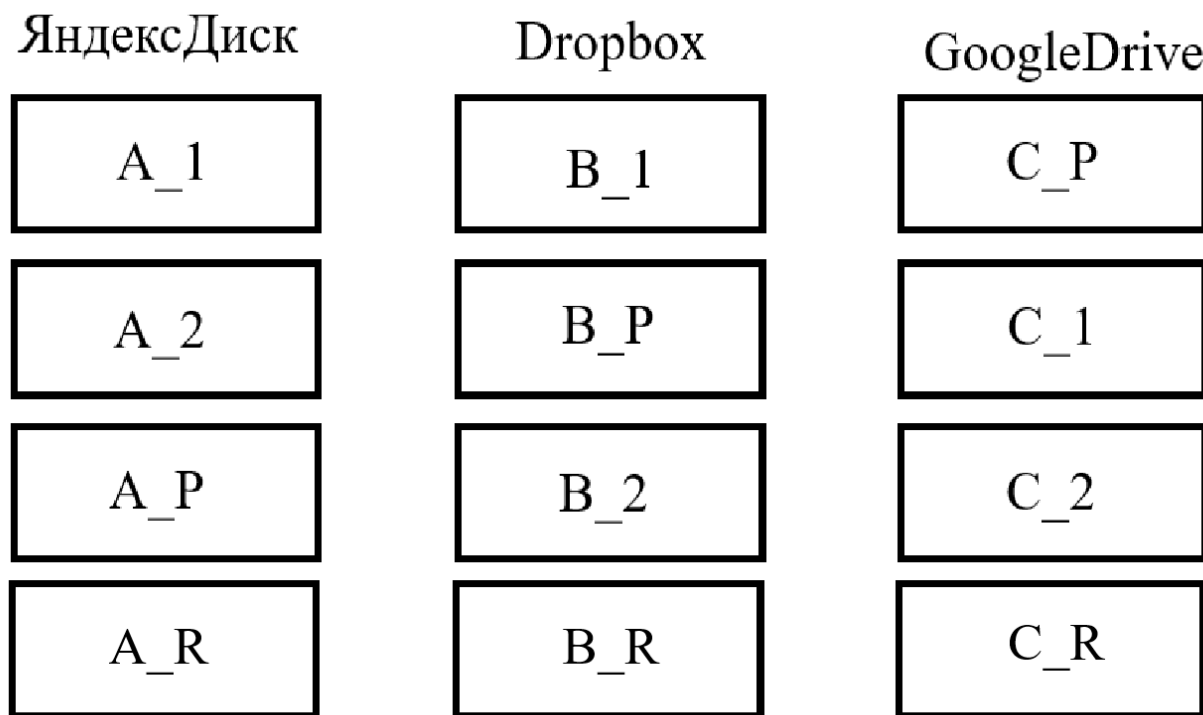


Рисунок 3 – Распределения данных среди облачных сервисов

По скольку в данном решении необходимо множество раз пересчитыв-

вать чётность для каждого массива байт, был реализован отдельный метод, принимающий на вход два набора данных. Данный алгоритм проходит по каждой паре элементов, производя логическую операцию $A \oplus B = Parity$, возвращая подготовленные данные:

```
public byte[] SolveParity(byte[] data_1, byte[] data_2)
{
    byte[] parity = new byte[data_1.Length];
    for(int i = 0; i < parity.Length; i++)
        parity[i] = (byte)(data_1[i] ^ data_2[i]);
    return parity;
}
```

5.1 Загрузка файла с распределением

Для распределения файла по принципу RAID 5 необходимо разбить его данные, представленные в виде набора байтов, на 6 равных частей. Это достигается путем считывания выбранного файла и записывания его содержимого в массив в виде набора байтов. Расчет размера всех массивов байтов производится путем деления общего количества байт на 6 с округлением вверх. Информация об образовавшихся в процессе округления лишних байтах сохраняется в отдельную переменную. Для разделения создаются 6 массивов и равномерно заполняются изначальными данными, в последнем массиве байты, превышающие изначальный размер файла записываются в виде нулей. Переменная *j* в данном случае является счетчиком для сохранения позиции считывания из изначального массива. Пример предоставлен в листинге ниже:

```
int size = (int)Math.Ceiling((double)fileContent.Length / 6);
int reduce = 6 - fileContent.Length % 6;
int j = 0;
byte[] data_1 = new byte[size];
for (int i = 0; i < data_1.Length; i++)
{
    data_1[i] = fileContent[j];
    j++;
}
```

Данные массивы преобразуются в виде файлов, хранящих в себе набор подготовленных байтов, они в свою очередь загружаются на диск. Далее был создан массив для хранения текущей четности пары данных, а также набор байт, хранящий в себе информацию о лишних байтах. Данные массивы заполняются при отправке каждой пары элементов. Для загрузки собранного файла из байтов необходимо его создать на сервере и после отправки удалить. Загрузка производится за счёт ранее реализованных сервисов для взаимодействия с облачными хранилищами. Пример предоставлен на листинге ниже:

```
byte[] parity = new byte[fileContent.Length / 6];
string dir = Directory.GetCurrentDirectory();
parity = SolveParity(data_1, data_2);
string fileName = Path.GetFileNameWithoutExtension(filePath);
string cur_fileName = fileName + "A_1";
await File.WriteAllBytesAsync(
    Path.Combine(dir, cur_fileName), data_1);
await _yandexdiskService.UploadFile(
    cur_fileName, Path.Combine(dir, cur_fileName));
File.Delete(Path.Combine(dir, cur_fileName));
cur_fileName = fileName + "B_1";
await File.WriteAllBytesAsync(
    Path.Combine(dir, cur_fileName), data_2);
await _dropboxService.UploadFile(
    Path.Combine(dir, cur_fileName), "/" + cur_fileName);
File.Delete(Path.Combine(dir, cur_fileName));
cur_fileName = fileName + "C_P";
await File.WriteAllBytesAsync(
    Path.Combine(dir, cur_fileName), parity);
```

Таким образом, производится загрузка 3 пар данных вместе с подготовленными для каждой из пар массивами четности. В конце метода производится обработка ответов от сервисов. В случае возникновения ошибок или конфликтов производится обработка их уведомлений.

5.2 Считывание данных

Скачивание данных с дисков производится за счёт поиска на них всех 6 частей, либо считывания чётности, в случае, если один из дисков в данный момент недоступен, а также загрузки файла, хранящего число лишних байт. Для загрузки данных создаётся массив, который будет хранить в себе набор байтов всех его частей. Загрузка производится за счёт подготовленных ранее сервисов, они возвращают набор байтов данного файла. Пример предоставлен на листинге ниже:

```
string curFileName = fileName + "A_1";
byte[] data_1 = await _yandexdiskService.DownloadFile(curFileName);
curFileName = fileName + "A_R";
byte[] reduce_bytes = await _yandexdiskService
    .DownloadFile(curFileName);
int reduce = BitConverter.ToInt32(reduce_bytes, 0);
byte[] data = new byte[data_1.Length * 6 - reduce];
int j = 0;
for (int i = 0; i < data_1.Length; i++)
{
    data[j] = data_1[i];
    j++;
}
```

Для обработки случаев, в которых один из дисков недоступен, подготовлено 3 метода, реализующих восстановление данных при помощи массивов чётности. Каждый метод реализует загрузку данных, в зависимости от того, какая информация была утрачена. В данном случае производится предварительная загрузка массива чётности для потерянных данных и производится операция $A \oplus Parity = B$ для восстановления недоступного элемента пары. Например, если недоступен Яндекс Диск и необходимо восстановить элемент A_2 производится операция $B_P \oplus C_1 = A_2$:

```
curFileName = fileName + "B_P";
byte[] data_3 = await _dropboxService.
    DownloadFile("/" + curFileName);
curFileName = fileName + "C_1";
```

```

await _googledriveService.DownloadFile(curFileName,
    Directory.GetCurrentDirectory() + "/" + curFileName);
byte[] data_4 = File.ReadAllBytes(
    Directory.GetCurrentDirectory() + "/" + curFileName);
data_3 = SolveParity(data_3, data_4);
for (int i = 0; i < data_3.Length; i++)
{
    data[j] = data_3[i];
    j++;
}
for (int i = 0; i < data_4.Length; i++)
{
    data[j] = data_4[i];
    j++;
}

```

6 Разработка пользовательского интерфейса

Пользовательский интерфейс был написан при помощи фреймворка `Vue.js`. Он имеет встроенные инструменты для разработки, такие как `Vue DevTools`, которые позволяют в реальном времени отслеживать состояние компонентов, их данные и события, что значительно упрощает процесс отладки.

Помимо вышеописанного фреймворка, основой одностраничного приложения стали технологии `HTML` и `CSS`. `HTML` (`HyperText Markup Language`) — это стандартный язык разметки, используемый для создания веб-страниц. Браузеры загружают `HTML`-документы с сервера используя протоколы `HTTP/HTTPS` либо с локального устройства, а затем преобразуют код в визуальный интерфейс, отображаемый на экране [16].

Основу `HTML` составляют элементы, которые формируют структуру страницы. С их помощью можно добавлять текстовые блоки, изображения, формы, ссылки и другие интерактивные компоненты. Элементы обозначаются тегами, заключёнными в угловые скобки. Хотя сами теги не отображаются в браузере, они определяют, как будет выглядеть и функционировать контент веб-страницы.

`CSS` (`Cascading Style Sheets`) — это язык стилей, который определяет внешний вид и оформление веб-документов, написанных на `HTML` или `XML` (включая `XHTML`, `SVG` и другие форматы). С помощью `CSS` можно задавать стили для элементов страницы, контролируя их отображение на экране, при печати, в голосовых интерфейсах и других медиа [17].

`CSS` работает по принципу каскадных правил, позволяя управлять макетом, шрифтами, цветами, анимацией и другими визуальными аспектами веб-страницы. В отличие от `HTML`, который отвечает за структуру контента, `CSS` сосредоточен на его презентации, обеспечивая разделение содержания и дизайна.

При разработке интерфейса был задействован `HTML` элемент `button`. Данный элемент может быть использован в любой части страницы, выполняет функцию кнопки, при нажатии на которую будет производиться заранее заготовленное событие. Её внешний вид может быть отредактирован при помощи `CSS`. Этот компонент имеет следующие поля для настройки [18]:

— `autofocus` — Этот атрибут определяет, получит ли кнопка фокус авто-

матически при загрузке страницы. Фокус останется на элементе, пока пользователь вручную не переключится на другой объект. В пределах одной формы только один элемент может иметь этот атрибут.

- `autocomplete` — Этот атрибут поддерживается только в браузере Firefox. В отличие от других браузеров, Firefox сохраняет состояние отключенной кнопки (установленное динамически) даже после перезагрузки страницы.
- `disabled` — Данный атрибут отключает взаимодействие пользователя с кнопкой. Когда ни у кнопки, ни у её контейнера нет атрибута `disabled`, элемент остаётся активным и доступным для действий.
- `form` — Это поле позволяет указать элемент, с которым связана кнопка.
- `formaction` — Определяет URL-адрес, на который будет отправлена форма при нажатии кнопки.
- `formenctype` — Определяет тип контента, отправляемого на сервер.
- `formmethod` — Определяет метод HTTP-запроса для отправки данных на сервер.
- `formnovalidate` — Указывает, что данные формы не будут валидироваться при отправке.
- `formtarget` — Указывает, где отображать ответ, полученный после отправки формы.
- `name` — Хранит название элемента.
- `type` — Устанавливает тип кнопки. Доступны следующие параметры:
 - `submit` — Кнопка отправляет данные формы на сервер.
 - `reset` — Кнопка сбрасывает все элементы управления к их начальным значениям.
 - `button` — Кнопка не имеет поведения по умолчанию.
 - `menu` — Кнопка открывает всплывающее меню.
- `value` — Исходное значение элемента.

В работе данный элемент применялся в нескольких местах. Например, кнопка, отвечающая за загрузку файла, вызывает соответствующий метод и имеет поле, делающее её недоступной до выбора пользователем файла:

```
<button
  @click="downloadFiles()"
  class="download-btn"
```

```
:disabled="!selectedServerFile"  
>  
Скачать  
</button>
```

На рисунке 4 продемонстрировано применение данного компонента на пользовательском интерфейсе в состоянии, когда он недоступен.



Рисунок 4 – Элемент `button` в неактивном состоянии с полем `disabled`

На рисунке 5 предоставлен элемент `button` при активном состоянии поля `disabled`.



Рисунок 5 – Элемент `button` в активном состоянии с полем `disabled`

`V-if` и `V-for` — это специальные директивы в Vue.js, которые используются для управления отображением элементов и итерации по массивам или объектам соответственно.

- `V-if` — это директива условного рендеринга, которая позволяет отображать или скрывать элементы на основе логического условия. Если условие истинно (`true`), элемент будет отрендерен, если ложно (`false`) — он не будет присутствовать в DOM [19].
- `V-for` — это директива для циклического рендеринга, которая позволяет итеративно отрисовывать элементы на основе массива, объекта или числа. Каждый элемент списка рендерится как отдельный DOM-элемент [20].

Компонент **V-if** в данной работе применялся для отображения элементов при соблюдении определённых условий. **V-for** позволяет отображать список элементов массива. Всплывающий список выбора элемента для загрузки отображается при условии нажатия соответствующей кнопки. При нажатии происходит переключение булевой переменной для отслеживания текущего состояния. Названия отображаются циклично, сортируясь по их индексу, при нажатии на название вызывается метод запоминающий выбранный файл и переключающий состояние отображения:

```
<button @click="showFileDropdown = !showFileDropdown"
class="select-btn">
    Выбрать файл для скачивания
</button>
<div v-if="showFileDropdown" class="dropdown">
    <div
        v-for="(file, index) in state.files"
        :key="index"
        class="dropdown-item"
        @click="selectServerFile(file)"
    >
        {{ file }}
    </div>
</div>
<div v-if="selectedServerFile" class="selected-file">
    Выбран файл: {{ selectedServerFile }}
</div>
```

Пример отображения данного элемента на пользовательском интерфейсе продемонстрирован на рисунке 6.

Выбрать файл для скачивания

image.jpg

presentation.pptx

4.png

Рисунок 6 – Всплывающее окно выбор файла

Элемент для выбора способа загрузки файла был реализован в виде `radio button`. Данный компонент позволяет выбирать пользователю один из предложенных вариантов пользователю. Для определения его параметров объявляется отдельный список:

```
const options = ref([
  { value: 'option1', label: 'Загрузка со всех дисков' },
  { value: 'option2', label: 'Загрузка без Яндекс Диска' },
  { value: 'option3', label: 'Загрузка без Dropbox' },
  { value: 'option4', label: 'Загрузка без Google Drive' }
]);
```

Отображение происходит при помощи элемента `V-for` сортируя по указанному в списке значению. Поле `type` отвечает за тип компонента, в данном случае это `radio button`, `v-model` позволяет синхронизировать данное значение с кодом JavaScript для применения его в остальных методах:

```
<label v-for="option in options"
      :key="option.value" class="radio-label">
```

```
<input
  type="radio"
  v-model="selectedOption"
  :value="option.value"
>
```

В зависимости от выбранного метода пользователь может восстановить утерянные данные с неработающего диска. При выборе опции со всеми дисками загрузка будет производиться быстрее за счёт отсутствия дополнительных операций для восстановления данных при помощи хранящихся массивов четности. Пример отображения данного элемента продемонстрирован на рисунке 7.

Выберите метод загрузки:

- ☒ **Загрузка со всех дисков**
- ☐ **Загрузка без Яндекс Диска**
- ☐ **Загрузка без Dropbox**
- ☐ **Загрузка без Google Drive**

Рисунок 7 – Окно выбора метода загрузки

В левой части экрана расположен список всех имеющихся в данный момент на облаке файлов. Данный список реализован посредством циклического отображения всех элементов. Вызов метода для получения набора

файлов производится при инициализации страницы. Он производится путем запроса данных с сервера:

```
const getFilesNames = async () => {  
  const response = await fetch(  
    'https://localhost:7229/api/GoogleDrive/files';  
  return await response.json();  
}
```

Пример отображения всех файлов продемонстрирован на рисунке 8.

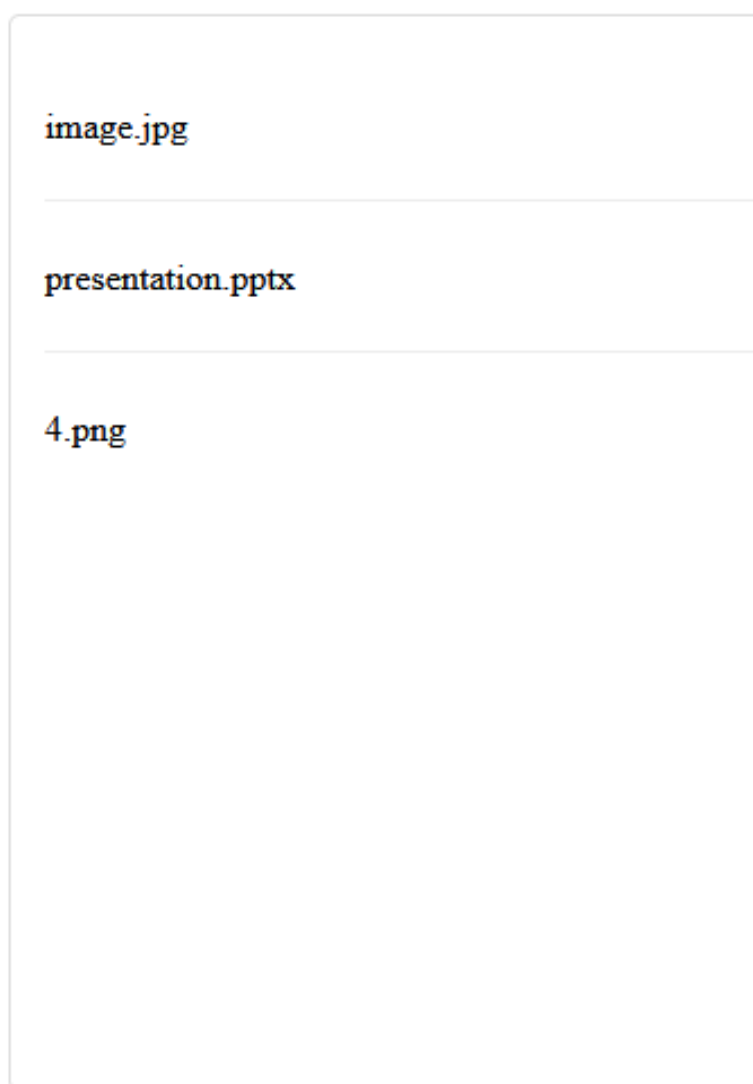


Рисунок 8 – Список доступных файлов

Метод скачивания файлов реализует запрос к back-end части приложения. Изначально производится проверка, был ли выбран файл для скачивания, в случае если нет, метод приостанавливает свою работу, уведомляя

пользователя о необходимости выбора. Метод загрузки определяется за счет значения, полученного от элемента `radio button`:

```
var responseUrl = ''
switch(selectedOption.value) {
  case 'option1':
    responseUrl = 'https://localhost:7229/api/RAID5/
download/${selectedServerFile.value}';
    break;
  case 'option2':
    responseUrl = 'https://localhost:7229/api/RAID5/
downloadwithoutya/${selectedServerFile.value}';
    break;
  case 'option3':
    responseUrl = 'https://localhost:7229/api/RAID5/
downloadwithoutgoogle/${selectedServerFile.value}';
    break;
  case 'option4':
    responseUrl = 'https://localhost:7229/api/RAID5/
downloadwithoutdropbox/${selectedServerFile.value}';
    break;
}
const response = await fetch(responseUrl);
```

При определении метода в отдельную переменную сохраняется адрес для проведения запроса. При помощи данной переменной производится запрос к контроллеру по сохраненному адресу с текущим значением пути к файлу. По завершении запроса начинается сборка файла и в последствии его скачивание для применения.

Загрузка файла на диски производится при помощи формирования `POST` запроса. При помощи параметров запроса в контроллер передаётся строка, отвечающая за местонахождение файла на физическом носителе. Также за счёт поля указывается, что форматом передаваемых данных является `JSON`:

```
const response = await fetch('https://localhost:7229/api/
```

```
RAID5/upload/${encodeURIComponent(selectedLocalFile.value.path)}',
{
    method: 'POST',
    headers: {
        'Content-Type': 'application/json'
    }
});
```

В случае возникновения конфликта во всплывающем окне отображается причина ошибки.

При загрузке файла picture.png на облачное хранилище он представляется в виде файлов, хранящих в себе распределённые по облачным сервисам наборы байт. Результат выполнения данной операции, в виде хранящихся данных на дисках, представлен на рисунке 9.

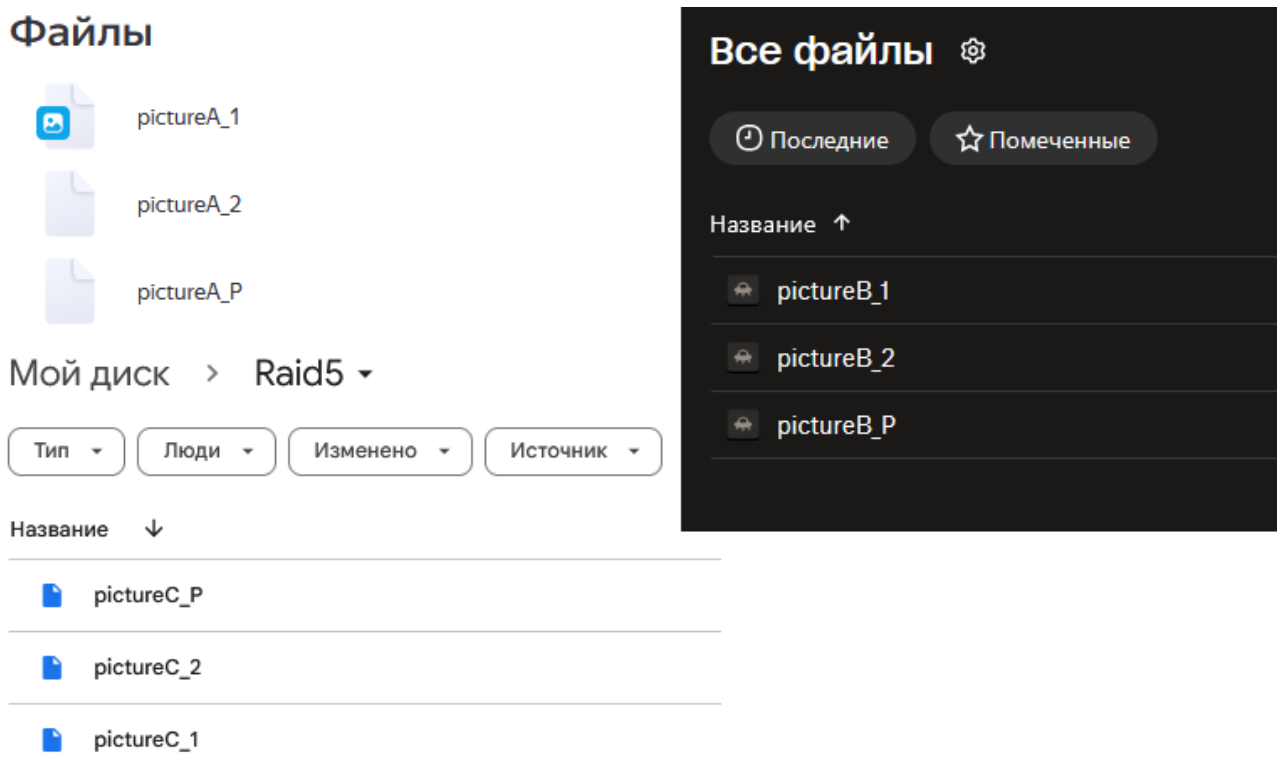


Рисунок 9 – Результат загрузки файла

Пример полного интерфейса для работы с приложением предоставлен на рисунке 10.

image.jpg

presentation.pptx

4.png

Выбрать файл для загрузки

Загрузить файл

Выбрать файл для скачивания

Скачать

Выберите метод загрузки:

☒ Загрузка со всех дисков

☐ Загрузка без Яндекс Диска

☐ Загрузка без Dropbox

☐ Загрузка без Google Drive

Рисунок 10 – Полная версия интерфейса

Полный код программной реализации предоставлен в Приложениях А, Б, В, Г, Д, Е, Ж, З.

7 Тестирование реализованного решения

Для проверки корректности работы алгоритма восстановления данных по принципу RAID 5 используется следующий подход:

1. Пользователь загружает файл F , который разбивается на блоки и распределяется между облачными хранилищами с вычислением контрольных блоков чётности.
2. В дальнейшем файл восстанавливается из двух блоков и одного блока чётности, находящихся в разных хранилищах.
3. Полученный восстановленный файл F' сравнивается с исходным F двумя способами:
 - побайтовое сравнение содержимого;
 - сравнение хеш-сумм:

$$\text{SHA256}(F) \stackrel{?}{=} \text{SHA256}(F').$$

4. При совпадении считается, что восстановление выполнено корректно.

Схема данного алгоритма представлена на рисунке 11.

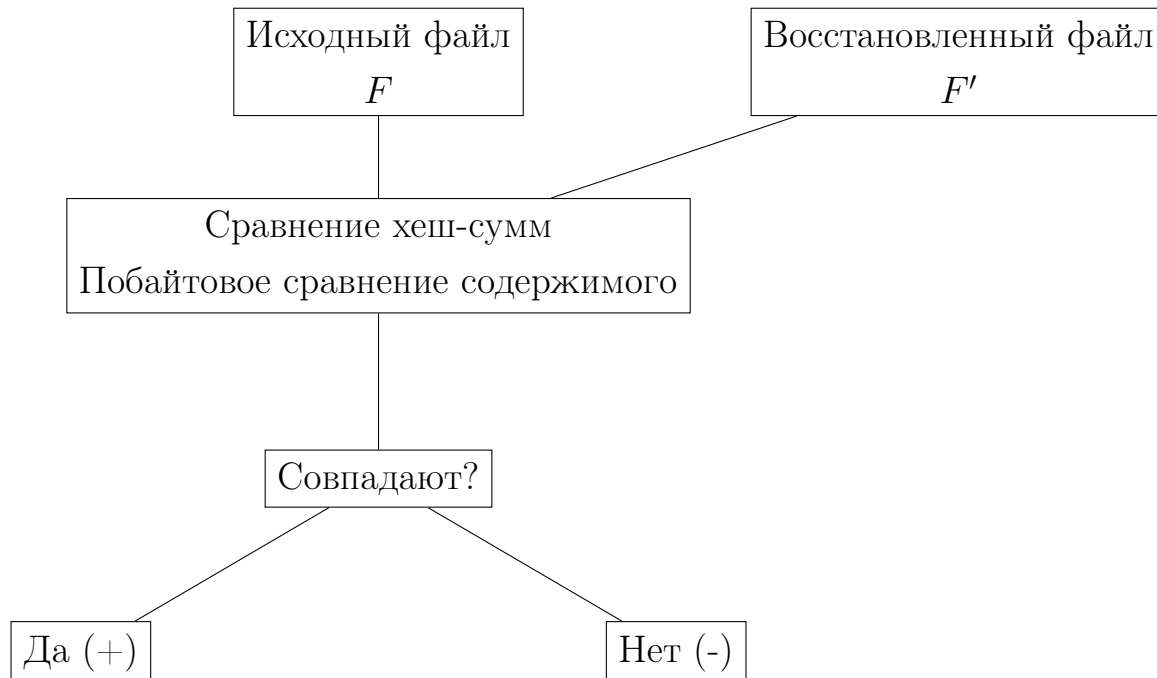


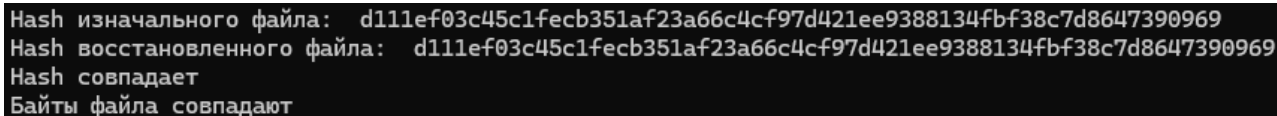
Рисунок 11 – Схема сравнения исходного и восстановленного файла

Для реализации данного тестирования была написана вспомогательная программа, производящая сравнение хеш-сумм, а также байтов двух выбранных файлов. Сравнение байтов производится путём представления файла

в виде массивов байтов и последующего сравнения значений. Для сравнения хеша создаётся экземпляр класса SHA256. Данный класс имеет метод для вычисления хеша файла, полученные значения сравниваются и результат сравнения выводится в консоль. Пример реализации метода вычисления хеш-суммы предоставлен на листинге ниже:

```
static string GetHash(string filePath)
{
    using var sha256 = SHA256.Create();
    using var stream = File.OpenRead(filePath);
    var hash = sha256.ComputeHash(stream);
    return BitConverter.ToString(hash)
        .Replace("-", "").ToLowerInvariant();
}
```

Тестирование было произведено при помощи сравнения изначального файла и файла, полученного после скачивания с применением разработанного решения. Результат данного теста представлен на рисунке 12.



```
Hash изначального файла: d111ef03c45c1fecb351af23a66c4cf97d421ee9388134fbf38c7d8647390969
Hash восстановленного файла: d111ef03c45c1fecb351af23a66c4cf97d421ee9388134fbf38c7d8647390969
Hash совпадает
Байты файла совпадают
```

Рисунок 12 – Результат сравнения изначального файла и восстановленного

Также было проведено тестирование производительности. Размер файла составляет 1,5 Мбайта на устройстве с процессором 12th Gen Intel(R) Core(TM) i7-12700H частота 2.30 МГц. Объем оперативной памяти 16 ГБайт с частотой 4800 МГц. По результатам тестирования прослеживается следующая динамика:

- Среднее время чтения — 5.6 с.
- Среднее время записи — 11.5 с.
- Среднее время восстановление данных Яндекс Диска — 7.3 с.
- Среднее время восстановление данных Google Drive — 7.3 с.
- Среднее время восстановление данных Dropbox — 4.4 с.

Результаты тестов демонстрируют стабильность работы системы: отклонения в времени выполнения операций незначительны (например, для записи — от 5.13 до 6.32 секунд). Восстановление данных при потере Google

Drive происходит быстрее, чем для других сервисов в силу того, что применённая библиотека затрачивает дополнительные ресурсы. Полученные в ходе тестирования значения предоставлены в таблице 2.

Таблица 2 – Результаты тестов производительности

№	Запись, с	Чтение, с	Восстановление Яндекс Диска, с	Восстановление Google Drive, с	Восстановление Dropbox, с
1	11.32	5.65	7.30	4.55	7.52
2	12.54	5.84	7.43	4.62	7.48
3	11.84	6.32	8.15	4.21	7.05
4	11.02	5.55	7.11	4.80	7.19
5	11.21	5.87	7.62	4.17	7.41
6	12.01	5.33	8.08	4.28	7.26
7	11.52	5.61	7.17	4.11	7.39
8	11.23	5.65	7.34	4.36	7.14
9	11.41	5.22	7.53	4.15	7.66
10	11.62	5.13	7.41	5.02	7.16

ЗАКЛЮЧЕНИЕ

В рамках данной бакалаврской работы было разработано решение, реализующее распределение файлов в виде RAID массива 5 уровня на облачных хранилищах. Приложение реализовано при помощи фреймворков ASP.NET Core, Vue 3, а также технологий HTML, CSS и JavaScript.

В процессе разработки было разработано решение для взаимодействия с облачными сервисами, рассмотрен принцип работы и реализации RAID5. Реализация для обработки данных представлена на ряде облачных хранилищ:

- Яндекс Диск,
- Dropbox,
- GoogleDrive.

Проведённые тесты также подтвердили корректность работы системы при внештатных ситуациях: система обеспечивает целостность файлов и позволяет восстанавливать их в случае недоступности одного из хранилищ.

Итоговое приложение представляет собой интерфейс для загрузки файлов на облачные хранилища с распределением по принципу RAID5. Также оно позволяет скачивать все файлы при помощи применения всех трех дисков, либо с учетом потери одного из них.

Разработанное приложение имеет потенциал к дальнейшему улучшению и практическому применению, так как позволяет хранить данные с повышенной отказоустойчивостью с сохранением преимуществ облачных сервисов.

Таким образом, в рамках работы создано решение, сочетающее преимущества облачного хранения и отказоустойчивости RAID 5, что делает его актуальным инструментом для безопасного и надёжного управления данными.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

- 1 Chen P.M., Lee E.K., Gibson G.A., Katz R.H., Patterson D.A. RAID: High-performance, reliable secondary storage / P.M. Chen, E.K. Lee, G.A. Gibson, R.H. Katz, D.A. Patterson. – ACM Computing Surveys (CSUR), 1994г. – 185 с.
- 2 Массивы дисков в RAID [Электронный ресурс] URL: <https://1bx.host/stati/obshchie-stati/massivy-diskov-v-raid0-raid1-raid5-raid10/> (дата обращения: 19.01.2025)
- 3 Павлова А.А. Получение доступа к данным, содержащимся в RAID 5 / Павлова А.А., Молодцова Ю.В. // Вестник Алтайской академии экономики и права. - 2022г. -№6-2. -с. 356-360.
- 4 Swagger Docs [Электронный ресурс] URL: <https://swagger.io/docs/> (дата обращения: 25.01.2025)
- 5 Cross-Origin Resource Sharing (CORS) [Электронный ресурс] URL: <https://developer.mozilla.org/ru/docs/Web/HTTP/Guides/CORS> (дата обращения: 03.02.2025)
- 6 ASP.NET Core [Электронный ресурс] URL: <https://learn.microsoft.com/ru-ru/aspnet/core/security/cors?view=aspnetcore-9.0> (дата обращения: 04.02.2025)
- 7 Яндекс ID [Электронный ресурс] URL: <https://yandex.ru/dev/id/doc/ru/register-client> (дата обращения: 13.02.2025)
- 8 Получение OAuth-токена [Электронный ресурс] URL: <https://yandex.ru/dev/id/doc/ru/access> (дата обращения: 13.02.2025)
- 9 Dropbox Documentation [Электронный ресурс] URL: https://www.dropbox.com/developers/documentation?_tk=pilot_lp&_ad=topbar1&_camp=docs (дата обращения: 20.02.2025)
- 10 Dropbox OAuth guide [Электронный ресурс] URL: <https://www.dropbox.com/lp/developers/reference/oauth-guide.html> (дата обращения: 22.02.2025)

- 11 NuGet Package Manager [Электронный ресурс] URL: <https://learn.microsoft.com/en-us/nuget/consume-packages/install-use-packages-visual-studio> (дата обращения: 02.03.2025)
- 12 OAuth 2.0 Scopes for Google APIs [Электронный ресурс] URL: <https://developers.google.com/identity/protocols/oauth2/scopes> (дата обращения: 04.03.2025)
- 13 Class UserCredential [Электронный ресурс] URL: <https://cloud.google.com/dotnet/docs/reference/Google.Apis/latest/Google.Apis.Auth.OAuth2.UserCredential> (дата обращения: 10.03.2025)
- 14 Class GoogleClientSecrets [Электронный ресурс] URL: <https://cloud.google.com/java/docs/reference/google-api-client/latest/com.google.api.client.googleapis.auth.oauth2.GoogleClientSecrets> (дата обращения: 12.03.2025)
- 15 Rendering Mechanism [Электронный ресурс] URL: <https://vuejs.org/guide/extras/rendering-mechanism> (дата обращения: 20.03.2025)
- 16 mdn web docs HTML [Электронный ресурс] URL: <https://developer.mozilla.org/ru/docs/Web/HTML> (дата обращения: 22.03.2025)
- 17 CSS: каскадные таблицы стилей [Электронный ресурс] URL: <https://developer.mozilla.org/ru/docs/Web/CSS> (дата обращения: 23.03.2025)
- 18 <button> - элемент кнопки [Электронный ресурс] URL: <https://developer.mozilla.org/ru/docs/Web/HTML/Reference/Elements/button> (дата обращения: 28.03.2025)
- 19 Conditional Rendering [Электронный ресурс] URL: <https://vuejs.org/guide/essentials/conditional.html> (дата обращения: 05.04.2025)
- 20 Отрисовка списков [Электронный ресурс] URL: <https://ru.vuejs.org/guide/essentials/list> (дата обращения: 05.04.2025)

ПРИЛОЖЕНИЕ А

Исходный код Program

```
using Microsoft.OpenApi.Models;
using vkr;

// Создание билдера приложения
var builder = WebApplication.CreateBuilder(args);

// Добавление сервисов для работы с MVC
builder.Services.AddControllersWithViews();
// Настройка генерации Swagger документации
builder.Services.AddSwaggerGen(options =>
{
    // Определение информации о API
    options.SwaggerDoc("v1", new OpenApiInfo
    {
        Version = "v1",           // Версия API
        Title = "RAID5Clouds",    // Название API
        Description = "Vkr",      // Описание API
    });
});
// Настройка политики CORS
builder.Services.AddCors(options =>
{
    // Создание политики с именем "AllowVueApp"
    options.AddPolicy("AllowVueApp",
        policy => policy.WithOrigins("http://localhost:5173")
        // Разрешенный источник
        .AllowAnyHeader()
        // Разрешение любых заголовков
        .AllowAnyMethod());
    // Разрешение любых HTTP-методов
});
```

```
// Вызов метода конфигурации сервисов из класса Startup
Startup.ConfigureServices(builder.Services);
// Сборка приложения
var app = builder.Build();

// Проверка среды выполнения
if (!app.Environment.IsDevelopment())
{
    // В production: использование страницы ошибок
    app.UseExceptionHandler("/Home/Error");
    // Включение HTTP Strict Transport Security
    app.UseHsts();
}
else
{
    // В development: включение Swagger middleware
    app.UseSwagger();
    // Настройка UI Swagger
    app.UseSwaggerUI(c => c.SwaggerEndpoint
        ("/swagger/v1/swagger.json", "vkr"));
}

// Перенаправление HTTP-запросов на HTTPS
app.UseHttpsRedirection();
// Поддержка статических файлов
app.UseStaticFiles();

// Применение политики CORS
app.UseCors("AllowVueApp");

// Включение маршрутизации
app.UseRouting();

// Включение авторизации
```

```
app.UseAuthorization();

// Настройка маршрутов по умолчанию
app.MapControllerRoute(
    name: "default", // Имя маршрута
    pattern: "{controller=Home}/{action=Index}/{id?}");
// Шаблон URL

// Запуск приложения
app.Run();
```

ПРИЛОЖЕНИЕ Б

Исходный код Startup

```
using vkr.Services;
namespace vkr
{
    // Статический класс для настройки сервисов приложения
    public static class Startup
    {
        // Метод для регистрации сервисов в DI-контейнере
        public static void ConfigureServices
            (IServiceCollection services)
        {
            // Добавление контроллеров с поддержкой сериализации
            //JSON
            services.AddControllers().AddNewtonsoftJson();

            // Регистрация HttpClient для выполнения HTTP-запросов
            services.AddHttpClient();

            // Регистрация сервиса для работы с Яндекс.Диском
            services.AddSingleton<YandexDiskService>();

            // Регистрация сервиса для работы с Dropbox
            services.AddSingleton<DropboxService>();

            // Регистрация сервиса для работы с Google Drive
            services.AddSingleton<GoogleDriveService>();

            // Регистрация сервиса RAID5
            services.AddSingleton<RAID5Service>();
        }
    }
}
```


ПРИЛОЖЕНИЕ В

Исходный код YandexDiskService

```
using System.Net.Http;
using System.Net.Http.Headers;
using System.Text;
using System.Threading.Tasks;
using Microsoft.Extensions.Configuration;
using Newtonsoft.Json;

namespace vkr.Services
{
    /// <summary>
    /// Сервис для работы с Яндекс диском
    /// Позволяет загружать и скачивать файлы
    /// </summary>
    public class YandexDiskService
    {
        private readonly HttpClient _httpClient;
        // клиент для отправки запросов
        private readonly string _apiUrl; // URL API Яндекс диска
        private readonly string _oAuthToken; // Токен авторизации
        /// <summary>
        /// Инициализация экземпляра класса
        /// </summary>
        /// <param name="configuration">Конфигурация,
        /// содержащая настройки для работы с API</param>
        public YandexDiskService(IConfiguration configuration)
        {
            _httpClient = new HttpClient();
            _apiUrl = configuration["YandexDisk:ApiUrl"];
            // URL API из конфигурации
            _oAuthToken = configuration["YandexDisk:OAuthToken"];
            // Токен из конфигурации
            _httpClient.DefaultRequestHeaders.Authorization = new
```

```

        AuthenticationHeaderValue("OAuth", _oauthToken);
        // Заголовок авторизации
    }
    /// <summary>
    /// Загрузка файла на Яндекс диск
    /// </summary>
    /// <param name="filePath">Путь к файлу на Яндекс диске
    </param>
    /// <param name="localFilePath">
    /// Содержимое файла в виде массива байтов</param>
    /// <returns>Результат загрузки</returns>
    public async Task<string> UploadFile
    (string filePath, string localFilePath)
    {
        var uploadUrlResponse = await _httpClient.GetAsync
        ($"{_apiUrl}resources/upload?path={filePath}
        &overwrite=true&fields=name,_embedded.items.path");
        // URL для загрузки файла
        uploadUrlResponse.EnsureSuccessStatusCode();
        // Статус запроса

        // Десериализация ответа с адресом для загрузки
        var uploadUrl = JsonConvert.DeserializeObject
        <YandexDiskUploadResponse>(await
        uploadUrlResponse.Content.ReadAsStringAsync()).Href;

        byte[] fileContent = File.ReadAllBytes(localFilePath);

        var content = new ByteArrayContent(fileContent);
        // Загрузка файла по полученному URL
        var uploadResponse = await _httpClient
        .PutAsync(uploadUrl, content);
        uploadResponse.EnsureSuccessStatusCode();
        // Проверка, что загрузка прошла успешно
    }

```

```

        return await uploadResponse.Content.ReadAsStringAsync();
        // Результат загрузки
    }

    /// <summary>
    /// Скачивание файла с Яндекс диска
    /// </summary>
    /// <param name="filePath">Путь к файлу на яндекс диске
    </param>
    /// <returns>Содержимое файла в виде массива байтов
    </returns>
    public async Task<byte[]> DownloadFile(string filePath)
    {

        var downloadUrlResponse = await _httpClient
            .GetAsync($"{_apiUrl}resources/download?path=
            {filePath}");
        // URL для загрузки файла
        downloadUrlResponse.EnsureSuccessStatusCode();
        // Статус запроса
        // Десериализация ответа с адресом для скачивания
        var downloadUrl = JsonConvert
            .DeserializeObject<YandexDiskDownloadResponse>
            (await downloadUrlResponse.Content
            .ReadAsStringAsync()).Href;

        var downloadResponse = await _httpClient
            .GetAsync(downloadUrl);
        // Скачивание файла по полученному URL
        downloadResponse.EnsureSuccessStatusCode();
        // Проверка результата запроса

        return await downloadResponse.Content
            .ReadAsByteArrayAsync();
    }

```

```

        // Содержимое файла
    }
    /// <summary>
    /// Класс для десериализации ответа API
    ///при получении URL для загрузки файла
    /// </summary>
    private class YandexDiskUploadResponse
    {
        public string Href { get; set; }
        // URL для загрузки файла
    }
    /// <summary>
    /// Класс для десериализации ответа API
    ///при получении URL для скачивания файла
    /// </summary>
    private class YandexDiskDownloadResponse
    {
        public string Href { get; set; }
        // URL для скачивания файла
    }
}
}

```

ПРИЛОЖЕНИЕ Г

Исходный код DropboxService

```
using System.Net.Http.Headers;
using Newtonsoft.Json;

namespace vkr.Services
{
    public class DropboxService
    {
        // HTTP-клиент для отправки запросов к Dropbox API
        private readonly HttpClient _httpClient;
        // Токен доступа для авторизации в Dropbox
        private readonly string _accessToken;
        // Базовый URL Dropbox API
        private readonly string _apiUrl;

        // Конструктор: инициализация сервиса с конфигурацией
        public DropboxService(IConfiguration configuration)
        {
            // Создание нового экземпляра HttpClient
            _httpClient = new HttpClient();
            // Получение токена доступа из конфигурации
            _accessToken = configuration["Dropbox:AccessToken"];
            // Получение базового URL API из конфигурации
            _apiUrl = configuration["Dropbox:ApiUrl"];
            // Установка заголовка авторизации (Bearer-токен)
            _httpClient.DefaultRequestHeaders.Authorization =
                new AuthenticationHeaderValue
                    ("Bearer", _accessToken);
        }

        // Метод для загрузки файла в Dropbox
        public async Task<string> UploadFile
            (string localFilePath, string dropboxPath)
```

```

{
    // Проверка существования локального файла
    if (!File.Exists(localFilePath))
    {
        throw new FileNotFoundException
            ("Файл не найден", localFilePath);
    }

    // Чтение содержимого файла в массив байтов
    byte[] fileContent = File.ReadAllBytes(localFilePath);

    // Создание контента запроса из байтов файла
    using (var content = new ByteArrayContent(fileContent))
    {
        // Установка типа контента
        content.Headers.ContentType =
            new MediaTypeHeaderValue
                ("application/octet-stream");

        // Создание HTTP-запроса
        var request = new HttpRequestMessage
            (HttpMethod.Post, $"{_apiUrl}upload");

        // Добавление специального заголовка
        request.Headers.Add("Dropbox-API-Arg",
            JsonConvert.SerializeObject(new
            {
                // Путь в Dropbox
                path = dropboxPath,
                // Режим перезаписи
                mode = "overwrite",
                // Автопереименование при конфликте
                autorename = true,
                // Уведомлять о действии
            }
        ));
    }
}

```

```

        mute = false
    }));

    // Установка контента запроса
    request.Content = content;

    // Отправка запроса и получение ответа
    var response = await _httpClient
        .SendAsync(request);
    // Проверка на успешность запроса
    response.EnsureSuccessStatusCode();

    // Возврат ответа в виде строки
    return await response.Content
        .ReadAsStringAsync();
}

}

// Метод для скачивания файла из Dropbox
public async Task<byte[]> DownloadFile(string dropboxPath)
{
    // Создание HTTP-запроса
    var request = new HttpRequestMessage
        (HttpMethod.Post, $"{_apiUrl}download");

    // Добавление заголовка с метаданными запроса
    request.Headers.Add("Dropbox-API-Arg",
        JsonConvert.SerializeObject(new
        {
            // Путь к файлу в Dropbox
            path = dropboxPath
        }));

    // Отправка запроса и получение ответа

```

```

var response = await _httpClient.SendAsync(request);
// Проверка на успешность запроса
response.EnsureSuccessStatusCode();

// Десериализация метаданных файла из заголовка ответа
var result = JsonConvert.DeserializeObject
    <DropboxFileMetadata>(
        response.Headers
            .GetValues("Dropbox-API-Result").First());

// Возврат содержимого файла в виде массива байтов
return await response.Content.ReadAsByteArrayAsync();
}

// Вложенный класс для десериализации метаданных
private class DropboxFileMetadata
{
    // Имя файла
    public string name { get; set; }
    // Нижний регистр пути
    public string path_lower { get; set; }
    // Размер файла в байтах
    public long size { get; set; }
}
}
}

```


ПРИЛОЖЕНИЕ Д

Исходный код GoogleDriveService

```
using System.Net.Http.Headers;
using System.Text.Json;
using System.Text;
using Google.Apis.Auth.OAuth2;
using Google.Apis.Drive.v3;
using Google.Apis.Services;
using Microsoft.AspNetCore.Http.HttpResults;
using System.Net;
using Google.Apis.Upload;
using System;

namespace vkr.Services
{
    public class GoogleDriveService
    {
        // Путь к файлу с учетными данными
        private readonly string credentialsPath;
        // ID папки в Google Drive
        private readonly string folderId;

        // Конструктор: инициализация сервиса
        public GoogleDriveService(IConfiguration configuration)
        {
            // Файл с OAuth2 credentials
            credentialsPath = "credentials.json";
            // ID целевой папки
            folderId = "1sxHqWNxEft-5gVvqM_WtdUG1ZQaPFsiB";
        }

        // Метод для загрузки файла в Google Drive
        public async Task<string> UploadFile(string localFilePath)
        {
```

```

// Проверка существования файла
if (!File.Exists(localFilePath))
    throw new FileNotFoundException
        ("File not found", localFilePath);
// Получение имени файла
var fileName = Path.GetFileName(localFilePath);
// Чтение файла
var fileContent = await File
    .ReadAllBytesAsync(localFilePath);

// Аутентификация в Google API
UserCredential credentials;
var clientSecrets = await GoogleClientSecrets
    .FromFileAsync(credentialsPath);

credentials = await GoogleWebAuthorizationBroker
    .AuthorizeAsync(
        clientSecrets.Secrets,
        // Запрашиваемые разрешения
        new[] { DriveService.ScopeConstants.DriveFile },
        "user",
        CancellationToken.None);

// Создание сервиса Google Drive
var service = new DriveService(new
    BaseClientService.Initializer()
{
    // Учетные данные
    HttpClientInitializer = credentials,
    // Имя приложения
    ApplicationName = "Raid5"
});

// Метаданные файла для загрузки

```

```

var fileMetaData = new Google.Apis.Drive.v3.Data.File()
{
    // Имя файла
    Name = Path.GetFileName(localFilePath),
    // Родительская папка
    Parents = new List<string> { folderId }
};

FilesResource.CreateMediaUpload request;

// Загрузка файла
using (var stream = new FileStream
    (localFilePath, FileMode.Open))
{
    request = service.Files.Create
        (fileMetaData, stream, "");
    request.Fields = "id"; // Запрашиваем только ID файла
    // Асинхронная загрузка
    var progress = await request.UploadAsync();

    if (progress.Status != UploadStatus.Completed)
        throw new Exception($"Upload failed:
            {progress.Exception?.Message}");
}
// Возвращаем имя загруженного файла
return request.ResponseBody.Name;
}

// Метод для скачивания файла из Google Drive
public async Task DownloadFile
    (string fileName, string localSavePath)
{
    // Аутентификация (аналогично UploadFile)
    UserCredential credentials;

```

```

var clientSecrets = await GoogleClientSecrets
    .FromFileAsync(credentialsPath);

credentials = await GoogleWebAuthorizationBroker
    .AuthorizeAsync(
        clientSecrets.Secrets,
        new[] { DriveService.ScopeConstants.DriveFile },
        "user",
        CancellationToken.None);

// Создание сервиса
var service = new DriveService
    (new BaseClientService.Initializer()
    {
        HttpClientInitializer = credentials,
        ApplicationName = "Raid5"
    });

// Поиск файла по имени
var fileId = await FindFileIdByName(fileName, service);
var fileInfo = await service.Files
    .Get(fileId).ExecuteAsync();

if (fileInfo == null)
    throw new Exception
        ("File not found in Google Drive");

// Скачивание файла
using (var fileStream = new FileStream
    (localSavePath, FileMode.Create, FileAccess.Write))
{
    var request = service.Files.Get(fileId);
    // Асинхронное скачивание
    await request.DownloadAsync(fileStream);
}

```

```

    }
}

// Метод для получения списка файлов в папке
public async Task<List<string>> GetFileNamesInFolder()
{
    // Аутентификация
    UserCredential credentials;
    var clientSecrets = await GoogleClientSecrets
        .FromFileAsync(credentialsPath);

    credentials = await GoogleWebAuthorizationBroker
        .AuthorizeAsync(
            clientSecrets.Secrets,
            new[] { DriveService.ScopeConstants.DriveFile },
            "user",
            CancellationToken.None);

    // Создание сервиса
    var service = new DriveService
        (new BaseClientService.Initializer()
        {
            HttpClientInitializer = credentials,
            ApplicationName = "Raid5"
        });

    // Запрос списка файлов
    var request = service.Files.List();
    // Фильтр: файлы в папке, не в корзине
    request.Q =
        $"'{folderId}' in parents and trashed = false";
    // Запрашиваем только имена файлов
    request.Fields = "files(name)";
}

```

```

        var result = await request.ExecuteAsync();
        // Возвращаем список имен
        return result.Files.Select(file => file.Name).ToList();
    }

    // Вспомогательный метод для поиска файла по имени
    public async Task<string> FindFileIdByName
    (string fileName, DriveService service)
    {
        var request = service.Files.List();
        // Фильтр по имени
        request.Q =
            $"name = '{fileName}' and trashed = false";
        // Запрашиваем ID и имя
        request.Fields = "files(id, name)";

        var result = await request.ExecuteAsync();
        // Возвращаем ID первого найденного файла
        return result.Files.FirstOrDefault()?.Id;
    }
}
}

```

ПРИЛОЖЕНИЕ E

Исходный код Raid5Service

```
using System.IO;
using vkr.Controllers;
using static System.Runtime.InteropServices.JavaScript.JSType;

namespace vkr.Services
{
    /// <summary>
    /// Сервис для реализации RAID 5
    /// </summary>
    public class RAID5Service
    {
        // Сервисы для работы с различными облачными хранилищами
        private readonly DropboxService _dropboxService;
        private readonly GoogleDriveService _googledriveService;
        private readonly YandexDiskService _yandexdiskService;

        /// <summary>
        /// Конструктор, инициализирующий сервисы облачных хранилищ
        /// </summary>
        public RAID5Service(DropboxService dropboxService,
            GoogleDriveService googledriveService,
            YandexDiskService yandexdiskService)
        {
            _dropboxService = dropboxService;
            _googledriveService = googledriveService;
            _yandexdiskService = yandexdiskService;
        }

        /// <summary>
        /// Метод для записи данных с использованием RAID 5
        /// Разделяет файл на части, вычисляет четность
        /// и распределяет по хранилищам
    }
}
```

```

/// </summary>
/// <param name="filePath">Путь к файлу для записи</param>
/// <returns>Результат операции</returns>
public async Task<string> WriteData(string filePath)
{
    string response = "Ok";
    // Чтение всего содержимого файла в массив байт
    byte[] fileContent = await File
        .ReadAllBytesAsync(filePath);
    // Размер каждой части файла с округлением вверх
    int size = (int)Math
        .Ceiling((double)fileContent.Length / 6);
    // Количество лишних байт
    int reduce = 6 - fileContent.Length % 6;
    // Разделение данных на 6 равных частей
    int j = 0;
    byte[] data_1 = new byte[size];
    for(int i = 0; i < data_1.Length; i++)
    {
        data_1[i] = fileContent[j];
        j++;
    }

    byte[] data_2 = new byte[size];
    for (int i = 0; i < data_2.Length; i++)
    {
        data_2[i] = fileContent[j];
        j++;
    }

    byte[] data_3 = new byte[size];
    for (int i = 0; i < data_3.Length; i++)
    {
        data_3[i] = fileContent[j];

```



```

        j++;
    }

    byte[] data_4 = new byte[size];
    for (int i = 0; i < data_4.Length; i++)
    {
        data_4[i] = fileContent[j];
        j++;
    }

    byte[] data_5 = new byte[size];
    for (int i = 0; i < data_5.Length; i++)
    {
        data_5[i] = fileContent[j];
        j++;
    }

    byte[] data_6 = new byte[size];
    // Последняя часть заполняется до конца данных
    // Все последующие байты представляются в виде 0
    for (int i = 0; i < data_6.Length; i++)
    {
        if (j < fileContent.Length)
        {
            data_6[i] = fileContent[j];
            j++;
        }
        else
        {
            data_6[i] = 0;
            j++;
        }
    }
}

```

```

// Массив для хранения битов четности
byte[] parity = new byte[fileContent.Length / 6];
string dir = Directory.GetCurrentDirectory();

try
{
    // Вычисление четности для первых двух блоков
    parity = SolveParity(data_1, data_2);

    // Формирование имени файла и
    // сохранение первого блока
    string fileName = Path
        .GetFileNameWithoutExtension(filePath);
    string cur_fileName = fileName + "A_1";
    await File.WriteAllBytesAsync
        (Path.Combine(dir, cur_fileName), data_1);

    // Загрузка в Yandex Disk и
    // удаление локального файла
    await _yandexdiskService.UploadFile
        (cur_fileName,
         Path.Combine(dir, cur_fileName));
    File.Delete(Path.Combine(dir, cur_fileName));

    // Сохранение и загрузка
    // второго блока в Dropbox
    cur_fileName = fileName + "B_1";
    await File.WriteAllBytesAsync
        (Path.Combine(dir, cur_fileName), data_2);
    await _dropboxService.UploadFile
        (Path.Combine(dir, cur_fileName),
         "/" + cur_fileName);
    File.Delete(Path.Combine(dir, cur_fileName));
}

```

```

// Сохранение и загрузка блока
// четности в Google Drive
cur_fileName = fileName + "C_P";
await File.WriteAllBytesAsync
    (Path.Combine(dir, cur_fileName), parity);
await _googledriveService
    .UploadFile(Path.Combine(
        dir, cur_fileName));
File.Delete(Path.Combine(dir, cur_fileName));

// Сохранение и загрузка третьего
// блока в Yandex Disk
cur_fileName = fileName + "A_2";
await File.WriteAllBytesAsync(
    Path.Combine(dir, cur_fileName), data_3);
await _yandexdiskService.UploadFile(
    cur_fileName, Path
        .Combine(dir, cur_fileName));
File.Delete(Path.Combine(dir, cur_fileName));

// Вычисление четности для третьего
// и четвертого блоков
parity = SolveParity(data_3, data_4);

// Сохранение и загрузка блока четности
// в Dropbox
cur_fileName = fileName + "B_P";
await File.WriteAllBytesAsync(
    Path.Combine(dir, cur_fileName), parity);
await _dropboxService.UploadFile(
    Path.Combine(dir, cur_fileName),
    "/" + cur_fileName);
File.Delete(Path.Combine(dir, cur_fileName));

```

```

// Сохранение и загрузка четвертого
// блока в Google Drive
cur_fileName = fileName + "C_1";
await File.WriteAllBytesAsync(
    Path.Combine(dir, cur_fileName), data_4);
await _googledriveService
    .UploadFile(Path.Combine(dir, cur_fileName));
File.Delete(Path.Combine(dir, cur_fileName));

// Сохранение и загрузка пятого блока в Dropbox
cur_fileName = fileName + "B_2";
await File.WriteAllBytesAsync(
    Path.Combine(dir, cur_fileName), data_5);
await _dropboxService.UploadFile(
    Path.Combine(dir, cur_fileName),
    "/" + cur_fileName);
File.Delete(Path.Combine(dir, cur_fileName));

// Сохранение и загрузка шестого
// блока в Google Drive
cur_fileName = fileName + "C_2";
await File.WriteAllBytesAsync(
    Path.Combine(dir, cur_fileName), data_6);
await _googledriveService
    .UploadFile(Path.Combine(dir, cur_fileName));
File.Delete(Path.Combine(dir, cur_fileName));

// Вычисление четности для пятого и шестого блоков
parity = SolveParity(data_5, data_6);

// Сохранение и загрузка блока
// четности в Yandex Disk
cur_fileName = fileName + "A_P";
await File.WriteAllBytesAsync

```

```

        (Path.Combine(dir, cur_fileName), parity);
await _yandexdiskService
    .UploadFile(cur_fileName,
        Path.Combine(dir, cur_fileName));
File.Delete(Path.Combine(dir, cur_fileName));
// Сохранение и загрузка числа
// излишних байт в Google Drive
byte[] bytes = BitConverter.GetBytes(reduce);
cur_fileName = fileName + "C_R";
await File.WriteAllBytesAsync
    (Path.Combine(dir, cur_fileName), bytes);

await _googledriveService.UploadFile
    (Path.Combine(dir, cur_fileName));
File.Delete(Path.Combine(dir, cur_fileName));
// Сохранение и загрузка числа
// излишних байт в Dropbox
cur_fileName = fileName + "B_R";
await File.WriteAllBytesAsync
    (Path.Combine(dir, cur_fileName), bytes);

await _dropboxService.UploadFile
    (Path.Combine(dir, cur_fileName)
        , "/" + cur_fileName);
File.Delete(Path.Combine(dir, cur_fileName));
// Сохранение и загрузка числа
// излишних байт в Yandex Disk
cur_fileName = fileName + "A_R";
await File.WriteAllBytesAsync
    (Path.Combine(dir, cur_fileName), bytes);

await _yandexdiskService.UploadFile
    (cur_fileName, Path.Combine
        (dir, cur_fileName));

```

```

        File.Delete(Path.Combine(dir, cur_fileName));

        return response;
    }
    catch(Exception ex)
    {
        // В случае ошибки возвращаем
        // сообщение об исключении
        return ex.Message;
    }
}

/// <summary>
/// Метод для чтения данных при доступности всех хранилищ
/// </summary>
/// <param name="fileName">Имя файла для чтения</param>
/// <returns>Массив байтов восстановленного файла</returns>
public async Task<byte[]> ReadData(string fileName)
{
    // Загрузка первого блока из Yandex Disk
    string curFileName = fileName + "A_1";
    byte[] data_1 = await _yandexdiskService
        .DownloadFile(curFileName);
    // Загрузка данных среза
    curFileName = fileName + "A_R";
    byte[] reduce_bytes = await _yandexdiskService
        .DownloadFile(curFileName);
    // Преобразование в число
    int reduce = BitConverter
        .ToInt32(reduce_bytes, 0);
    // Создание массива для полных данных
    byte[] data = new byte[data_1.Length * 6 - reduce];
    int j = 0;

```

```

// Копирование первого блока в результат
for(int i = 0; i < data_1.Length; i++)
{
    data[j] = data_1[i];
    j++;
}

// Загрузка и копирование второго блока из Dropbox
curFileName = fileName + "B_1";
byte[] data_2 = await _dropboxService
    .DownloadFile("/" + curFileName);
for (int i = 0; i < data_2.Length; i++)
{
    data[j] = data_2[i];
    j++;
}

// Загрузка и копирование третьего блока
// из Yandex Disk
curFileName = fileName + "A_2";
byte[] data_3 = await _yandexdiskService
    .DownloadFile(curFileName);
for (int i = 0; i < data_3.Length; i++)
{
    data[j] = data_3[i];
    j++;
}

// Загрузка и копирование четвертого блока Google Drive
curFileName = fileName + "C_1";
await _googledriveService.DownloadFile(curFileName,
    Directory.GetCurrentDirectory()
        + "/" + curFileName);
byte[] data_4 = File.ReadAllBytes(Directory

```

```

        .GetCurrentDirectory() + "/" + curFileName);
for (int i = 0; i < data_4.Length; i++)
{
    data[j] = data_4[i];
    j++;
}

// Загрузка и копирование пятого блока из Dropbox
curFileName = fileName + "B_2";
byte[] data_5 = await _dropboxService
    .DownloadFile("/" + curFileName);
for (int i = 0; i < data_5.Length; i++)
{
    data[j] = data_5[i];
    j++;
}

// Загрузка и копирование шестого блока
// из Google Drive
curFileName = fileName + "C_2";
await _googledriveService.DownloadFile(curFileName,
    Directory.GetCurrentDirectory() +
        "/" + curFileName);
byte[] data_6 = File.ReadAllBytes(
    Directory.GetCurrentDirectory()
        + "/" + curFileName);
for (int i = 0; i < data_6.Length - reduce; i++)
{
    data[j] = data_6[i];
    j++;
}

return data;
}

```



```

/// <summary>
/// Метод для восстановления данных при
/// недоступности Yandex Disk
/// Использует блоки четности из других хранилищ
/// </summary>
public async Task<byte[]> ReadDataWithoutYandex
    (string fileName)
{
    // Загрузка блока четности из Google Drive
    string curFileName = fileName + "C_P";
    await _googledriveService.DownloadFile(curFileName,
        Directory.GetCurrentDirectory() +
            "/" + curFileName);
    byte[] data_1 = File.ReadAllBytes(Directory
        .GetCurrentDirectory() + "/" + curFileName);
    // Загрузка данных среза
    curFileName = fileName + "C_R";
    await _googledriveService.DownloadFile(curFileName,
        Directory.GetCurrentDirectory() + "/"
            + curFileName);
    byte[] reduce_bytes = File.ReadAllBytes(
        Directory.GetCurrentDirectory() + "/"
            + curFileName);
    // Преобразование в число
    int reduce = BitConverter.ToInt32(reduce_bytes, 0);

    // Загрузка второго блока из Dropbox
    curFileName = fileName + "B_1";
    byte[] data_2 = await _dropboxService
        .DownloadFile "/" + curFileName);

    // Восстановление первого блока с использованием
    // четности

```

```

data_1 = SolveParity(data_1, data_2);

// Создание массива для полных данных
byte[] data = new byte[data_1.Length * 6 - reduce];
int j = 0;

// Копирование восстановленного первого блока
for (int i = 0; i < data_1.Length; i++)
{
    data[j] = data_1[i];
    j++;
}

// Копирование второго блока
for (int i = 0; i < data_2.Length; i++)
{
    data[j] = data_2[i];
    j++;
}

// Загрузка блока четности из Dropbox
curFileName = fileName + "B_P";
byte[] data_3 = await _dropboxService
    .DownloadFile("/" + curFileName);

// Загрузка четвертого блока из Google Drive
curFileName = fileName + "C_1";
await _googledriveService.DownloadFile(curFileName,
    Directory.GetCurrentDirectory() +
        "/" + curFileName);
byte[] data_4 = File.ReadAllBytes(Directory
    .GetCurrentDirectory() + "/" + curFileName);

// Восстановление третьего блока с

```

```

// использованием четности
data_3 = SolveParity(data_3, data_4);

// Копирование восстановленного третьего блока
for (int i = 0; i < data_3.Length; i++)
{
    data[j] = data_3[i];
    j++;
}

// Копирование четвертого блока
for (int i = 0; i < data_4.Length; i++)
{
    data[j] = data_4[i];
    j++;
}

// Загрузка пятого блока из Dropbox
curFileName = fileName + "B_2";
byte[] data_5 = await _dropboxService
    .DownloadFile("/") + curFileName);
for (int i = 0; i < data_5.Length; i++)
{
    data[j] = data_5[i];
    j++;
}

// Загрузка шестого блока из Google Drive
curFileName = fileName + "C_2";
await _googledriveService.DownloadFile(curFileName,
    Directory.GetCurrentDirectory() +
        "/" + curFileName);
byte[] data_6 = File.ReadAllBytes(Directory
    .GetCurrentDirectory() + "/" + curFileName);

```

```

        for (int i = 0; i < data_6.Length - reduce; i++)
        {
            data[j] = data_6[i];
            j++;
        }

        return data;
    }

    /// <summary>
    /// Метод для восстановления данных при
    /// недоступности Google Drive
    /// Использует блоки четности из других хранилищ
    /// </summary>
    public async Task<byte[]> ReadDataWithoutGoogle
        (string fileName)
    {
        // Загрузка первого блока из Yandex Disk
        string curFileName = fileName + "A_1";
        byte[] data_1 = await _yandexdiskService
            .DownloadFile(curFileName);
        // Загрузка данных среза
        curFileName = fileName + "B_R";
        byte[] reduce_bytes = await _dropboxService
            .DownloadFile("/") + curFileName);
        // Преобразование в число
        int reduce = BitConverter.ToInt32(reduce_bytes, 0);

        byte[] data = new byte[data_1.Length * 6 - reduce];
        int j = 0;

        // Копирование первого блока
        for (int i = 0; i < data_1.Length; i++)
        {

```

```

        data[j] = data_1[i];
        j++;
    }

    // Загрузка второго блока из Dropbox
    curFileName = fileName + "B_1";
    byte[] data_2 = await _dropboxService
        .DownloadFile("/") + curFileName);
    for (int i = 0; i < data_2.Length; i++)
    {
        data[j] = data_2[i];
        j++;
    }

    // Загрузка третьего блока из Yandex Disk
    curFileName = fileName + "A_2";
    byte[] data_3 = await _yandexdiskService
        .DownloadFile(curFileName);

    // Загрузка блока четности из Dropbox
    curFileName = fileName + "B_P";
    byte[] data_4 = await _dropboxService
        .DownloadFile("/") + curFileName);

    // Восстановление четвертого блока
    // с использованием четности
    data_4 = SolveParity(data_3, data_4);

    // Копирование третьего блока
    for (int i = 0; i < data_3.Length; i++)
    {
        data[j] = data_3[i];
        j++;
    }

```

```

// Копирование восстановленного четвертого блока
for (int i = 0; i < data_4.Length; i++)
{
    data[j] = data_4[i];
    j++;
}

// Загрузка пятого блока из Dropbox
curFileName = fileName + "B_2";
byte[] data_5 = await _dropboxService
    .DownloadFile("/") + curFileName);

// Загрузка блока четности из Yandex Disk
curFileName = fileName + "A_P";
byte[] data_6 = await _yandexdiskService
    .DownloadFile(curFileName);

// Восстановление шестого блока с
// использованием четности
data_6 = SolveParity(data_5, data_6);

// Копирование пятого блока
for (int i = 0; i < data_5.Length; i++)
{
    data[j] = data_5[i];
    j++;
}

// Копирование восстановленного шестого блока
for (int i = 0; i < data_6.Length - reduce; i++)
{
    data[j] = data_6[i];
    j++;
}

```

```

    }

    return data;
}

/// <summary>
/// Метод для восстановления данных при
/// недоступности Dropbox
/// Использует блоки четности из других хранилищ
/// </summary>
public async Task<byte[]> ReadDataWithoutDropbox
    (string fileName)
{
    // Загрузка первого блока из Yandex Disk
    string curFileName = fileName + "A_1";
    byte[] data_1 = await _yandexdiskService
        .DownloadFile(curFileName);
    // Загрузка данных среза
    curFileName = fileName + "A_R";
    byte[] reduce_bytes = await _yandexdiskService
        .DownloadFile(curFileName);
    // Преобразование в число
    int reduce = BitConverter.ToInt32(reduce_bytes, 0);

    byte[] data = new byte[data_1.Length * 6 - reduce];

    // Загрузка блока четности из Google Drive
    curFileName = fileName + "C_P";
    await _googledriveService.DownloadFile(curFileName,
        Directory.GetCurrentDirectory() +
            "/" + curFileName);
    byte[] data_2 = File.ReadAllBytes(Directory
        .GetCurrentDirectory() + "/" + curFileName);

```

```

// Восстановление второго блока с использованием
// четности
data_2 = SolveParity(data_1, data_2);
int j = 0;

// Копирование первого блока
for (int i = 0; i < data_1.Length; i++)
{
    data[j] = data_1[i];
    j++;
}

// Копирование восстановленного второго блока
for (int i = 0; i < data_2.Length; i++)
{
    data[j] = data_2[i];
    j++;
}

// Загрузка третьего блока из Yandex Disk
curFileName = fileName + "A_2";
byte[] data_3 = await _yandexdiskService
    .DownloadFile(curFileName);
for (int i = 0; i < data_3.Length; i++)
{
    data[j] = data_3[i];
    j++;
}

// Загрузка четвертого блока из Google Drive
curFileName = fileName + "C_1";
await _googledriveService.DownloadFile(curFileName,
    Directory.GetCurrentDirectory()
    + "/" + curFileName);

```



```

byte[] data_4 = File.ReadAllBytes(Directory
    .GetCurrentDirectory() + "/" + curFileName);
for (int i = 0; i < data_4.Length; i++)
{
    data[j] = data_4[i];
    j++;
}

// Загрузка блока четности из Yandex Disk
curFileName = fileName + "A_P";
byte[] data_5 = await _yandexdiskService
    .DownloadFile(curFileName);

// Загрузка шестого блока из Google Drive
curFileName = fileName + "C_2";
await _googledriveService.DownloadFile(curFileName,
    Directory.GetCurrentDirectory() +
        "/" + curFileName);
byte[] data_6 = File.ReadAllBytes(Directory
    .GetCurrentDirectory() + "/" + curFileName);

// Восстановление пятого блока
// с использованием четности
data_5 = SolveParity(data_5, data_6);

// Копирование восстановленного пятого блока
for (int i = 0; i < data_5.Length; i++)
{
    data[j] = data_5[i];
    j++;
}

// Копирование шестого блока
for (int i = 0; i < data_6.Length - reduce; i++)

```

```

        {
            data[j] = data_6[i];
            j++;
        }

        return data;
    }

    /// <summary>
    /// Метод для вычисления блока четности с
    /// использованием XOR
    /// </summary>
    /// <param name="data_1">Первый блок данных</param>
    /// <param name="data_2">Второй блок данных</param>
    /// <returns>Блок четности</returns>
    public byte[] SolveParity(byte[] data_1, byte[] data_2)
    {
        byte[] parity = new byte[data_1.Length];
        for(int i = 0; i < parity.Length; i++)
            // Побитовое XOR
            parity[i] = (byte)(data_1[i] ^ data_2[i]);
        return parity;
    }
}

```

ПРИЛОЖЕНИЕ Ж

Исходный код Raid5Controller

```
using Microsoft.AspNetCore.Mvc;
using vkr.Services;

namespace vkr.Controllers
{
    /// <summary>
    /// Контроллер для работы с RAID 5 - реализует API
    /// для записи и чтения данных
    /// с распределением по облачным хранилищам и возможностью
    /// восстановления при отказе одного хранилища
    /// </summary>
    [ApiController]
    // Базовый маршрут для всех методов контроллера
    [Route("api/[controller]")]
    public class RAID5Controller : Controller
    {
        // Сервис, реализующий логику RAID 5
        public readonly RAID5Service _raid5Service;

        /// <summary>
        /// Конструктор контроллера с внедрением
        /// зависимости RAID5Service
        /// </summary>
        /// <param name="raid5Service">
        ///     Сервис работы с RAID 5</param>
        public RAID5Controller(RAID5Service raid5Service)
        {
            _raid5Service = raid5Service;
        }

        /// <summary>
        /// Метод для записи файла в RAID 5
```

```

/// (разделение на части и распределение по хранилищам)
/// </summary>
/// <param name="filePath">Путь к файлу для записи</param>
/// <returns>Результат операции</returns>
[HttpPost("upload/{filePath}")]
public async Task<IActionResult> WriteData(string filePath)
{
    // Вызов сервиса для записи данных
    var result = await _raid5Service.WriteData(filePath);
    // Возвращаем результат операции
    return Ok(result);
}

/// <summary>
/// Метод для чтения файла из RAID 5
/// (при доступности всех хранилищ)
/// </summary>
/// <param name="fileName">Имя файла для чтения
/// (без расширения)</param>
/// <returns>Файл в виде потока байтов</returns>
[HttpGet("download/{fileName}")]
public async Task<IActionResult> ReadData(string fileName)
{
    // Вызов сервиса для чтения данных
    var fileContent = await _raid5Service.ReadData(fileName);
    // Возвращаем файл как поток байтов
    return File(fileContent, "application/octet-stream");
}

/// <summary>
/// Метод для чтения файла при недоступности Yandex Disk
/// (восстановление данных с использованием блоков четности)
/// </summary>
[HttpGet("downloadwithoutya/{fileName}")]

```

```

public async Task<IActionResult>
    ReadDataWithoutYandex(string fileName)
{
    // Вызов сервиса для чтения без Yandex Disk
    var fileContent = await _raid5Service
        .ReadDataWithoutYandex(fileName);
    return File(fileContent, "application/octet-stream");
}

/// <summary>
/// Метод для чтения файла при недоступности Google Drive
/// (восстановление данных с использованием блоков четности)
/// </summary>
[HttpGet("downloadwithoutgoogle/{fileName}")]
public async Task<IActionResult>
    ReadDataWithoutGoogle(string fileName)
{
    // Вызов сервиса для чтения без Google Drive
    var fileContent = await _raid5Service
        .ReadDataWithoutGoogle(fileName);
    return File(fileContent, "application/octet-stream");
}

/// <summary>
/// Метод для чтения файла при недоступности Dropbox
/// (восстановление данных с использованием блоков четности)
/// </summary>
[HttpGet("downloadwithoutdropbox/{fileName}")]
public async Task<IActionResult>
    ReadDataWithoutDropbox(string fileName)
{
    // Вызов сервиса для чтения без Dropbox
    var fileContent = await _raid5Service
        .ReadDataWithoutDropbox(fileName);

```

```
        return File(fileContent, "application/octet-stream");
    }
}
}
```

ПРИЛОЖЕНИЕ 3

Исходный код App.vue

```
<template>
  <div class="container">
    <!-- Список доступных файлов -->
    <div class="files-list">
      <div v-for="(file, index) in state.files"
        :key="index" class="file-item">
        <p>{{ file }}</p>
      </div>
    </div>

    <!-- Панель управления -->
    <div class="controls">
      <!-- Кнопка выбора файла для загрузки -->
      <button @click="openFilePicker" class="select-btn">
        Выбрать файл для загрузки
      </button>

      <!-- Скрытый input для выбора файла -->
      <input
        type="file"
        ref="fileInput"
        @change="handleFileSelect"
        style="display: none"
      >

      <!-- Отображение выбранного файла -->
      <div v-if="selectedLocalFile" class="selected-file">
        Название файла: {{ selectedLocalFile.name }}
      </div>

      <!-- Кнопка загрузки файла -->
      <button
        @click="sendFilePath"
        class="upload-btn"
        :disabled="!selectedLocalFile"
      >
```

```

>
    Загрузить файл
</button>

<!-- Выбор файла для скачивания -->
<div class="file-selection">
    <button @click="showFileDropdown =
        !showFileDropdown" class="select-btn">
        Выбрать файл для скачивания
    </button>
    <!-- Выпадающий список файлов -->
    <div v-if="showFileDropdown" class="dropdown">
        <div
            v-for="(file, index) in state.files"
            :key="index"
            class="dropdown-item"
            @click="selectServerFile(file)"
        >
            {{ file }}
        </div>
    </div>
    <!-- Отображение выбранного файла -->
    <div v-if="selectedServerFile"
        class="selected-file">
        Выбран файл: {{ selectedServerFile }}
    </div>
</div>

<!-- Кнопка скачивания -->
<button
    @click="downloadFiles()"
    class="download-btn"
    :disabled="!selectedServerFile"
>

```



```

        Скачать
    </button>

    <!-- Радио-кнопки выбора метода загрузки -->
    <div class="radio-group">
        <h3>Выберите метод загрузки:</h3>
        <label v-for="option in options"
            :key="option.value" class="radio-label">
            <input
                type="radio"
                v-model="selectedOption"
                :value="option.value"
            >
            {{ option.label }}
        </label>
    </div>
</div>
</div>
</template>

<script setup>
import { ref, onMounted } from 'vue';

// Состояние компонента
const state = ref({
    files: [] // Список доступных файлов
});

// Опции для выбора метода загрузки
const options = ref([
    { value: 'option1', label: 'Загрузка со всех дисков' },
    { value: 'option2', label: 'Загрузка без Яндекс Диска' },
    { value: 'option3', label: 'Загрузка без Dropbox' },
    { value: 'option4', label: 'Загрузка без Google Drive' }

```

```

]);

// Выбранный метод загрузки
const selectedOption = ref('option1');
// Выбранный файл для скачивания
const selectedServerFile = ref(null);
// Выбранный файл для загрузки
const selectedLocalFile = ref(null);
// Видимость выпадающего списка
const showFileDropdown = ref(false);
// Ссылка на input для выбора файла
const fileInput = ref(null);

/**
 * Получает список файлов с сервера
 */
const getFilesNames = async () => {
  const response = await fetch
    ('https://localhost:7229/api/GoogleDrive/files');
  return await response.json();
}

/**
 * Выбирает файл из списка для скачивания
 */
const selectServerFile = (fileName) => {
  selectedServerFile.value = fileName;
  showFileDropdown.value = false;
}

/**
 * Открывает диалог выбора файла
 */
const openFilePicker = () => {

```

```

    fileInput.value.click();
}

/**
 * Обрабатывает выбор файла
 */
const handleFileSelect = (event) => {
    const file = event.target.files[0];
    if (file) {
        selectedLocalFile.value = {
            name: file.name,
            path: file.path,
            size: file.size,
            type: file.type,
            lastModified: file.lastModified,
            file: file
        };
        console.log('Selected file object:',
            selectedLocalFile.value);
    }
}

/**
 * Скачивает выбранный файл с сервера
 */
const downloadFiles = async () => {
    if (!selectedServerFile.value) return;

    // Формируем URL в зависимости от выбранного метода
    var responseUrl = ''
    switch(selectedOption.value) {
        case 'option1':
            responseUrl = 'https://localhost:7229/api/
                RAID5/download/${selectedServerFile.value}';

```

```

        break;
    case 'option2':
        responseUrl = 'https://localhost:7229/
            api/RAID5/downloadwithoutya/${selectedServerFile.value}';
        break;
    case 'option3':
        responseUrl = 'https://localhost:7229/
            api/RAID5/downloadwithoutgoogle/
                ${selectedServerFile.value}';
        break;
    case 'option4':
        responseUrl = 'https://localhost:7229/
            api/RAID5/downloadwithoutdropbox/
                ${selectedServerFile.value}';
        break;
}

// Загружаем файл
const response = await fetch(responseUrl);
console.log(response)
const blob = await response.blob();

// Иницилируем скачивание
computeDownload(blob)
return await response;
}

/**
 * Иницирует скачивание файла
 */
function computeDownload(blob) {
    const url = window.URL.createObjectURL(blob);
    const link = document.createElement('a');
    link.href = url;

```

```

link.setAttribute('download', selectedServerFile.value);
document.body.appendChild(link);
link.click();
document.body.removeChild(link);
window.URL.revokeObjectURL(url);
}

/**
 * Отправляет путь к файлу на сервер для загрузки
 */
const sendFilePath = async () => {
  if (!selectedLocalFile.value?.path) {
    alert('Please select a file first');
    return;
  }

  try {
    const response = await fetch('https://localhost:7229/
      api/RAID5/upload/${encodeURIComponent(
        selectedLocalFile.value.path)}', {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json'
      }
    });

    if (!response.ok) {
      throw new Error('HTTP error!
        status: ${response.status}');
    }

    const result = await response.json();
    console.log('Path sent successfully:', result);
  }

```

```

    } catch (error) {
        console.error('Error sending file path:', error);
        alert('Error: ' + error.message);
    }
};

// При монтировании компонента загружаем список файлов
onMounted(async () => {
    state.value.files = await getFilesNames();
    console.log(state.value.files);
});
</script>

<style scoped>
/* Основные стили контейнера */
.container {
    display: flex;
    gap: 20px;
    padding: 20px;
}

/* Стили списка файлов */
.files-list {
    flex: 1;
    border: 1px solid #ddd;
    padding: 15px;
    border-radius: 5px;
}

.file-item {
    padding: 8px 0;
    border-bottom: 1px solid #eee;
}

```

```

.file-item:last-child {
    border-bottom: none;
}

/* Стили панели управления */
.controls {
    width: 250px;
    display: flex;
    flex-direction: column;
    gap: 20px;
}

.file-selection {
    position: relative;
}

/* Стили кнопок */
.select-btn {
    padding: 10px 15px;
    background-color: #2196F3;
    color: white;
    border: none;
    border-radius: 4px;
    cursor: pointer;
    font-size: 16px;
    width: 100%;
    margin-bottom: 8px;
}

.select-btn:hover {
    background-color: #0b7dda;
}

/* Стили выпадающего списка */

```

```

.dropdown {
  position: absolute;
  top: 100%;
  left: 0;
  right: 0;
  background: white;
  border: 1px solid #ddd;
  border-radius: 4px;
  box-shadow: 0 2px 5px rgba(0,0,0,0.2);
  z-index: 10;
  max-height: 200px;
  overflow-y: auto;
}

.dropdown-item {
  padding: 8px 15px;
  cursor: pointer;
}

.dropdown-item:hover {
  background-color: #f5f5f5;
}

/* Стили для отображения выбранного файла */
.selected-file {
  margin-top: 8px;
  padding: 8px;
  background: #f5f5f5;
  border-radius: 4px;
  font-size: 14px;
  margin-bottom: 12px;
  word-break: break-all;
}

```



```

/* Стили кнопки скачивания */
.download-btn {
  padding: 10px 15px;
  background-color: #4CAF50;
  color: white;
  border: none;
  border-radius: 4px;
  cursor: pointer;
  font-size: 16px;
}

.download-btn:hover {
  background-color: #45a049;
}

.download-btn:disabled {
  background-color: #cccccc;
  cursor: not-allowed;
}

/* Стили группы радио-кнопок */
.radio-group {
  display: flex;
  flex-direction: column;
  gap: 10px;
  padding: 15px;
  border: 1px solid #ddd;
  border-radius: 5px;
}

.radio-group h3 {
  margin: 0 0 10px 0;
}

```

```
.radio-label {
  display: flex;
  align-items: center;
  gap: 8px;
  cursor: pointer;
}

.radio-label input {
  cursor: pointer;
}

/* Стили кнопки загрузки */
.upload-btn {
  padding: 10px 15px;
  background-color: #ff9800;
  color: white;
  border: none;
  border-radius: 4px;
  cursor: pointer;
  font-size: 16px;
  margin-bottom: 8px;
}

.upload-btn:hover {
  background-color: #e68a00;
}

.upload-btn:disabled {
  background-color: #cccccc;
  cursor: not-allowed;
}
</style>
```