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 РЕЙД.

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 Теоретические сведения 6
1.1 Общие сведения о $RAID$ 6
1.2 $RAID$ 5 9
2 Разработка back-end 11
2.1 Общие сведения 11
2.2 Яндекс Диск 12
2.3 Dropbox 16
2.4 GoogleDrive 16
ЗАКЛЮЧЕНИЕ 16
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 16

ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

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

$RAID$ 5 — 5 уровень $RAID$;

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.

Целью данной бакалаврской работы является разработка приложения для хранения файлов на облачных сервисах с распределением по принципу $RAID5$. Данное приложение должно представлять собой удобный инструмент, позволяющий пользователю удаленно взаимодействовать с хранимыми данными.

В рамках работы предполагается изучение способов автоматического взаимодействия с облачными сервисами, а также метода организации файлов по принципу $RAID5$. Основой проекта выбрана технология $ASP.NETCore$, язык программирования $C\#$. Интерфейс представлен в виде одностраничного сайта с использованием языка $JavaScript$.

Данное приложение на данный момент разработано в целях предоставления пользователю удалённого доступа к файлам с повышенной отказоустойчивостью при их хранении. В дальнейшем при развитии данного проекта пользователю может быть предоставлена возможность расширения тарифного плана на облачных сервисах, а также автоматического обновления токенов доступа к дискам.

Результаты данной работы могут быть полезны для практических задач, связанных с применением отказоустойчивых методов для хранения данных на облачных сервисах. Для достижения цели были поставлены следующие задачи:

1. Изучить принцип работы $RAID5$;
2. Разработка решения для обработки данных на YandexDisk;
3. Разработка решения для обработки данных на GoogleDrive;
4. Разработка решения для обработки данных на Dropbox;
5. Реализовать модуль распределения данных по $RAID5$;

6. Реализовать пользовательский интерфейс.

1 Теоретические сведения

1.1 Общие сведения о $RAID$

$RAID$-контроллер предлагает более широкие возможности по сравнению с JBOD (просто набором дисков). Технология $RAID$ появилась в эпоху, когда жесткие диски были значительно дороже и менее надежными, чем сейчас. Изначально аббревиатура $RAID$ расшифровывалась как "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.

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.

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 уменьшается 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.

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 экономным по занимаемому месту, резервируя только одно

из предоставленных хранилищ данных, а также предоставляет исчерпывающую отказоустойчивость, позволяя восстановить данные при потере доступа к одному из дисков.

1.2 *RAID* 5

RAID 5 уровня является отказоустойчивым массивом хранения данных на нескольких хранилищах данных, в данном случае на облачных сервисах, представляющий 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. 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. Задается 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 $A_i, B_i, C_i, D_i, E_i, F_i$. 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. Пример такого распределения предоставлен на рисунке 1.

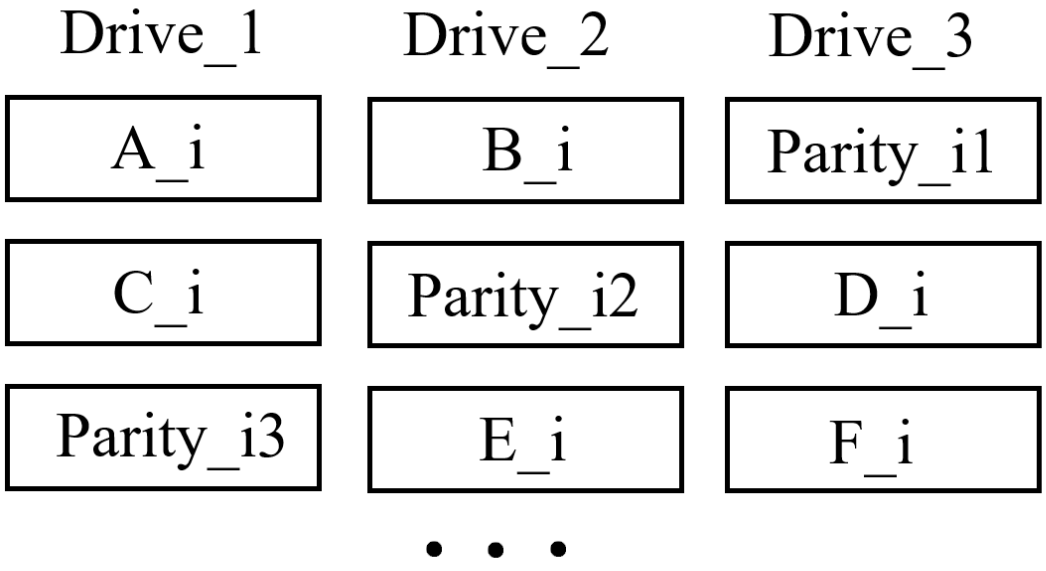

Рисунок 1 – Пример распределения данных

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.

2 Разработка back-end

2.1 Общие сведения

Для разработки серверной части проекта был выбран фрэймворк ASP.Net Core. ASP.NET Core представляет технологию для создания веб-приложений на платформе .NET, развиваемую компанией Microsoft. В качестве языка программирования для разработки приложений на ASP.NET Core использован C#.

Для разработки приложения был создан следующий набор классов:

— Program — отвечает за сборку проекта, в нем присутствует настройка CORS, Swagger, вызов сборки сервисов проекта.

— Startup — выполняет инициализацию представленных в проекте сервисов

— Набор сервисов, отвечающих за внутреннюю логику, выполняющие HTTP запросы

— Контроллеры для взаимодействия с внешними приложениями

Swagger представляет собой набор инструментов для тестирования и визуализации REST API. REST API — это конкретный API, который используется для обмена данными между клиентами и серверами в интернете. Основная часть данного инструмента при работе с REST API — это предоставление 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 инструментарий предоставляет возможность создания описаний работы API: информации о ресурсах, параметрах запросов, возвращаемых данных, конечных точках и других важных вещах. Чтобы автоматизировать это описание, сделать его структурированным и прозрачным.

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.

Например, если HTML-страница с сервера http://domain-a.com запрашивает изображение <img src="http://domain-b.com/image.jpg», 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 (XMLHttpRequest или Fetch API), следуя политике одинакового источника (same-origin policy). 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. Пример использования продемонстрирован на листинге ниже:

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

2.2 Яндекс Диск

Яндекс Диск является первым выбранным облачным хранилищем 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 на 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 создается вспомогательная страница для приема токена.

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 следствии чего к сервису необходимо подключить REST API, значительно расширяющий набор предоставляемых данных. Для реализации данного приложения были выбраны пункты чтение всего диска и запись 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. 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 экземпляр класса 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 токена, пути к файлу на облачном сервисе и массива байтов файла для отправки. Изначально запрашивается ссылка для загрузки файла. Полученный ответ хранится в JSON формате, из-за чего его требуется десериализовать. Далее формируется массив байтов с файлом и вместе с полученной ранее ссылкой формируется запрос для выгрузки. Пример данного кода приведен 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.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
}
```

Следующим шагом является скачивание файла с облачного сервиса. В тело запроса передаётся OAuth токен, путь к файлу, расположенном в хранилище. 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
}
```

2.3 Dropbox

2.4 GoogleDrive

ЗАКЛЮЧЕНИЕ

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ