Санкт–Петербургский государственный университет
Факультет математики и компьютерных наук

Вадим Маратович Салаватов

Выпускная квалификационная работа

Реализация мультиплатформенного доступа к файловым хранилищам на языке Kotlin

Уровень образования: бакалавриат
Направление 01.03.02 «Прикладная математика и информатика»
Основная образовательная программа СВ.5005.2018 «Прикладная математика, фундаментальная информатика и программирование»
Профиль «Современное программирование»

Научный руководитель:
профессор, д.ф.-м.н. А. С. Куликов

Консультант:
(TODO:) В. Н. Брагилевский

Рецензент:
девелопер-адвокат ООО «ИнтеллиДжей Лабс»
А. А. Архипов

Санкт-Петербург
2022 г.

Содержание

Введение

Современный рынок разработки ИТ-продуктов чрезвычайно конкурентен. От скорости создания минимального жизнеспособного продукта (MVP) может зависеть положение на этом рынке и пользовательский охват. В последнее время популярность набирают языки и инструменты, позволяющие создавать приложения из единой кодовой базы под множество целевых платформ: начиная от настольных компьютеров под управлением операционных систем Windows, Linux и Mac OS X, и заканчивая умными часами (watchOS) или умными телевизорами (tvOS). Такими, например, являются фреймворк Flutter, который написан на языке Dart, а также Kotlin Multiplatform, который является инструментом языка Kotlin. Оба этих инструмента относительно новые — первые публичные версии стали доступны в 2017 году, — и поэтому имеют недостаток в виде отсутствия большого набора готовых библиотек для разных нужд. Одним из недостатков подобных инструментов является также ограниченность доступной из общего кода функциональности, поскольку вся такая функциональность должна иметь реализацию на всех целевых платформах.

Примером такой функциональности является работа с файлами и файловыми системами. В частности, для Kotlin Multiplatform нет стандартного способа работы с файловыми хранилищами из общего кода, а значительным недостатком многих существующих решений является отсутствие поддержки браузерной платформы. Запрос от сообщества разработчиков на подобные решения подтверждается вопросами по этой теме на популярных интернет-площадках[1, 2, 3, 4, 5]. В данной работе изучается изложенная проблема и презентуется решение в виде мультиплатформенной Kotlin-библиотеки, предоставляющее доступ к файловым хранилищам и поддерживающее в том числе браузерную платформу.

Структура работы. В разделе 1 осуществлен разбор предметной области и альтернативных решений данной проблемы. В разделе 2 представлена реализованная мультиплатформенная библиотека. В разделе 3 рассказывается о технических деталях её реализации. В разделе 4 представлены примеры приложений, созданных на базе данной библиотеки.

Постановка задачи

Цель работы состоит в разработке мультиплатформенной библиотеки на языке программирования Kotlin, позволяющей разработчику описывать логику работы с различными файловыми хранилищами в общем модуле мультиплатформенного проекта. Библиотека должна предоставлять интерфейс виртуальной файловой системы с иерархической структурой папок и файлов и позволять записывать и читать файлы как массивы байт. Библиотека должна быть достаточно гибкой, чтобы разработчик мог самостоятельно дополнить её функциональность (например, поддержать на базе библиотеки новое хранилище), а также иметь возможность расширения на уровне предоставляемых интерфейсов, когда от целевых хранилищ требуется поддержка особых возможностей (например, наличие у файлов атрибутов прав на чтение/запись).

От конечного продукта ожидается как минимум:

- поддержка трех платформ: JVM (для приложений, работающих в среде операционных систем Windows, Linux, и т.п.), JS (браузерные приложения), Android (мобильные приложения);
- поддержка как минимум одного облачного хранилища, доступного со всех поддерживаемых платформ.

1. Обзор

1.1. Предметная область

Kotlin Multiplatform. Kotlin позволяет писать мультиплатформенные библиотеки и приложения с помощью Kotlin Multiplatform[6] — это особенность языка, благодаря которой можно переиспользовать модули кода между платформами, а также использовать платформоспецифичную функциональность из общего кода с помощью механизма `expect/actual`-объявлений.

Целевые платформы JS (JavaScript). Компилятор Kotlin/JS[7] позволяет транслировать Kotlin-код в JavaScript-код, предназначенный для исполнения, в основном, в двух средах: исполнение в браузере, когда коду доступны API веб-браузера и функциональность для управления содержимым веб-страницы, и исполнение на стороне сервера — в данном случае доступна функциональность Node.js, которая, например, предоставляет доступ к файловой системе операционной системы[8]. Важно понимать отличие этих двух вариантов, поскольку иногда мультиплатформенные проекты заявляют поддержку JavaScript как платформы, но не уточняют, в какой именно среде они могут работать. Далее в тексте данные целевые платформы будет обозначаться как JS (browser) и JS (node.js) соответственно.

Вирутальная файловая система. В данной работе под виртуальной файловой системой[9] (ВФС) подразумевается абстрактный интерфейс, предоставляющий функциональность обычной файловой системы, но конкретная реализация которого зависит от используемого файлового хранилища.

(TODO: что-то ещё?)

1.2. Существующие решения

Для поиска существующих решений были сделаны следующие запросы в поисковую систему Google:

- «kotlin multiplatform io»

- «kotlin multiplatform filesystem»
- «kotlin multiplatform file»
- «kotlin multiplatform storage»

Аналогичные запросы были сделаны и на русском языке, но никаких дополнительных результатов это не принесло.

Также был произведен поиск по релевантным теме вопросам на портале StackOverflow[10]. Несколько релевантных вопросов нашлось[1, 2], но дополнительных результатов они не принесли. Дополнительно был произведен поиск по репозиториям на портале GitHub[11]. Кроме решений, перечисленных ниже, был обнаружен курируемый сообществом репозиторий со списком Kotlin Multiplatform библиотек[12].

Список полученных релевантных решений можно разделить на три категории.

1.2.1 Решения, предоставляющие интерфейс виртуальных файловых систем

- **kile**[13]

  Последнее изменение кода — июль 2020 г., сайт с документацией недоступен. Из кода ясно, что (возможно) поддерживаются платформы JVM, JS (Node.js). Из особенностей: нет методов для чтения и записи файлов; конкретные хранилища подключаются через адаптеры; есть адаптер для FTP[14]-сессии, некоторые методы не реализованы, что-то потенциально работающее написано только для платформы JVM; есть адаптер для локальной файловой системы (аналогично реализация есть только для JVM).

- **files**[15]

  Последнее изменение кода — ноябрь 2020 г., в репозитории есть минимальный сопровождающий текст. Из кода ясно, что (возможно) поддерживаются платформы JVM, Android, JS (browser). Предоставляется мультиплатформенный интерфейс файла, но только с возможностью

чтения. Судя по коду, данное решение позволяет запрашивать у пользователя загрузку файла в веб-браузере.

- **kotlinx-fs**[16]

  Последнее изменение кода — февраль 2019 г., в репозитории есть минимальный сопровождающий текст. Заявлена поддержка JVM, JS (Node.js) и POSIX-совместимых операционных систем (Mac OS X и Linux) для нативных Kotlin-приложений. Предоставляется интерфейс локальной файловой системы (проверка существования пути, чтение атрибутов путей, создание файлов и папок, в том числе временных, перемещение и удаление путей, чтение и запись файлов) и его реализации на перечисленных платформах.

- **supernatural-fs**[17]

  Последнее изменение кода — декабрь 2020 г., есть минимальная документация. Предоставляется мультиплатформенный интерфейс локальной файловой системы, доступный на платформах Android, iOS, JS (Node.js), JVM.

- **korio**[18]

  Последнее изменение — август 2021 г., есть документация. Поддерживает очень много платформ: помимо Android, JVM, JS (browser), ещё и iOS, Mac OS X, Linux x64, watchOS, tvOS и другие. Библиотека является частью большого проекта (множества библиотек) Korlibs[19], берущего начало в 2017 году и направленного на создание мультиплатформенного движка для видеоигр и сопутствующих мультиплатформенных библиотек. В связи с этим, данные библиотеки сильно завязаны друг на друга, и, вероятно, разрабатывались для написания приложений именно в среде этого набора библиотек.

  Библиотека представляет свои реализации TCP- и HTTP-клиентов, примитивы для работы с потоковой обработкой данных (`AsyncStream`), собственную сериализацию/десериализацию форматов JSON, YAML, XML.

Есть абстракция виртуальной файловой системы `Vfs`, элементы иерархии — `VfsFile`, то есть нет различия на уровне типов между папками и файлами, но есть методы для определения этого (`isDirectory()`), поддерживаются атрибуты файлов, наблюдение за событиями файловой системы (`watch(path: String, handler: (FileEvent) -> Unit)`). Есть методы для чтения и записи файлов как целиком, так и в потоковом режиме (используя `AsyncStream`).

Среди доступных реализаций `Vfs` можно отметить `ZipVfs` — виртуальную файловую систему (ВФС) для чтения zip-архивов, `UrlVfs` — ВФС для чтения и записи ресурсов, расположенных в сети Интернет, по протоколу HTTP. Чтение доступно в потоковом режиме, запись — только целиком; реализация предполагает, что данные можно записать, отправив PUT-запрос с содержимым на удаленный сервер. Есть также `LocalVfs` для работы с локальным файловым хранилищем с несколькими реализациями для разных платформ.

- **okio**[20]

  Проект активно развивается, доступна документация. Является мультиплатформенным проектом, предоставляет в основном инструменты для мультиплатформенного ввода-вывода и потоковой обработки данных, однако доступна функциональность для работы с файловой системой через класс `FileSystem`. Главная проблема заключается в том, что для платформы JS требуется работа именно на базе библиотеки Node.js, то есть как серверное приложение, а не браузерное.

1.2.2 Key-value решения

Здесь перечислены найденные продукты с открытым исходным кодом, которые представляют собой мультиплатформенные библиотеки, позволяющие персистентно сохранять пары ключ-значение в локальном хранилище. Они не решают поставленную задачу или ее части, однако в некоторых случаях их функциональности может оказаться достаточно, поэтому они стоят упоминания.

Многие из этих проектов в качестве хранилищ используют `SharedPreferences` для Android, `window.localStorage` для JS (browser), `NSUserDefaults` для iOS.

- **multiplatform-settings**[21]

  Проект активно развивается, доступна документация. Помимо платформ JVM, JS (browser) и Android поддержаны также iOS, macOS, watchOS, tvOS и Windows. Предоставляет интерфейс для сохранения пар ключ-значение в локальном хранилище.

- **KVault**[22]

  Проект активно развивается, есть минимальная документация. Поддержаны только платформы iOS и Android, однако реализована поддержка шифрования данных.

- **Kissme**[23]

  Последнее изменение — январь 2020 г., есть минимальная документация. Аналогично предыдущему пункту, поддержаны платформы iOS и Android, реализовано шифрование данных.

- **multiplatform-preferences**[24]

  Проект заархивирован, последнее изменение — февраль 2020 г., есть минимальная документация. Поддержаны платформы iOS и Android.

- **Kotlin Data Storage**[25]

  Последнее изменение — октябрь 2021 г., есть минимальная документация. Поддержаны платформы JVM, JS (Node.js и browser), Android. Решение позволяет сохранять сериализуемые данные в локальных хранилищах (в т.ч. в файлы на платформе JVM). Благодаря использованию делегации свойств[26], работа с данными происходит в виде чтения и записи значений переменных.

- **asoft-storage**[27]

Последнее изменение кода — апрель 2020 г., нет документации и примеров. Из кода ясно, что (возможно) поддерживаются платформы JVM, Android, JS (browser). Не реализована логика работы на JVM.

- **kached**[28]

  Последнее изменение кода — октябрь 2020 г., есть небольшой сопровождающий текст. Из кода ясно, что (возможно) поддерживаются платформы JVM, Android, iOS. Для JVM реализовано хранилище на базе файловой системы. Из особенностей: поддерживается шифрование значений, хранение данных в Amazon S3[29] (но доступно только на платформе JVM).

1.2.3 Решения для мультиплатформенного ввода-вывода (IO)

Здесь перечислены решения, предоставляющие возможности для мультиплатформенного ввода-вывода данных. На их базе гипотетически можно построить решение главной задачи.

- **korio**[18]

  См. раздел 1.2.1.

- **okio**[20]

  См. раздел 1.2.1

- **kotlinx-io**[30]

  Официальная библиотека для мультиплатформенного ввода-вывода. Последнее изменение кода — май 2020 г., в вопросе об актуальности библиотеки[31] был дан ответ, что в разработке новая версия этой библиотеки. По всей видимости, проект стал частью Ktor[32] и развивается внутри него.

- **tinlok**[33]

  Последнее изменение кода — март 2022 г., доступна документация. Проект предоставляет средства ввода-вывода для нативных целей (Linux

AMD64/ARM64, Windows64 и другие): BSD-сокеты, пути для работы с файловой системой, буферы для работы с массивами байт, криптографические методы и другое.

2. Презентация решения

В данной главе дан высокоуровневый обзор реализованной библиотеки `multifs`[34], а также обоснования принятых архитектурных решений.

2.1. Архитектура библиотеки

Ввиду того, что одна из главных задач заключается в организации поддержки файловых хранилищ на платформе JS (browser), то есть в веб-браузерах, возникает вопрос, какие файловые хранилища вообще можно на этой платформе поддержать? Поскольку в современных веб-браузерах много внимания уделяется безопасности, в них отсутствует доступ к файловой системе операционной системы, на которой они запущены. Для сохранения данных между сеансами можно пользоваться свойством `localStorage`[35] — оно позволяет сохранять данные в виде пар ключ-значение. Гипотетически можно эмулировать файловую систему через эту структуру данных. Однако объем `localStorage` часто ограничен веб-браузерами и невелик: например, веб-браузер Firefox версии 98.0 ограничивает объем `localStorage` примерно десятью мегабайтами памяти. Если представить, что мы пишем, скажем, онлайн-редактор документов в формате docx и pdf, поддерживающее к тому же картинки, то сохранить много документов в таком хранилище просто не выйдет.

Необходим доступ к «настоящему» файловому хранилищу, и в таком случае можно предложить следующие альтернативы локальной файловой системе, доступные из веб-браузеров:

- облачные хранилища, предоставляющие HTTP REST[36] API интерфейс для доступа к данным (Google Drive, Yandex.Disk, Dropbox, Amazon S3, и т.д.);
- подключение к удаленным файловым системам посредством FTP[14], попытка реализации чего была предпринята в проекте kile[13], или другим протоколам передачи файлов (SFTP, WebDAV, и т.д.).

Поскольку целью работы является также создание как можно более

гибкой библиотеки в плане расширения множества поддерживаемых файловых хранилищ, имеет смысл выяснение функциональности и особенностей, которые имеют современные файловые хранилища.

2.1.1 Обзор функциональности и особенностей файловых хранилищ

Для изучения выбраны несколько наиболее популярных онлайн-сервисов для хранения данных с публичным API: Google Drive, Яндекс.Диск, Dropbox, Amazon S3. Здесь перечисляются наиболее интересная (по мнению автора) функциональность и особенности, которые могут пригодится при решении поставленной задачи, более подробная информация содержится в официальной документации сервисов.

Google Drive[37]. Файлы и папки не отличаются с точки зрения представления (то есть всё является файлом), различаются полем MimeType (у папок он равен `application/vnd.google-apps.folder`). Каждый файл имеет уникальный идентификатор (ID), при этом не запрещается иметь несколько файлов с одинаковым именем в одной папке. Файлы могут помимо своего содержимого хранить произвольные свойства (пары ключ-значение), есть атрибуты времени создания, модификации и доступа, атрибуты прав доступа. Файлы могут принадлежать одному из трёх изолированных пространств (spaces): пространство диска (drive space) — основное пространство, в котором пользователь хранит свои файлы; пространство приложения (app data folder space) — пространство, доступное только приложению и недоступное пользователю; пространство фотографий (photos space). Среди методов доступны создание файлов, скачивание, загрузка (в том числе больших файлов по частям), копирование (не папок), получение и изменение метаданных отдельно от содержимого (можно таким образом перемещать файлы), получение списка файлов с указанием фильтров (например, по id родителя).

Яндекс.Диск[38]. Строгая иерархия каталогов и файлов, имена файлов уникальны и чувствительны к регистру. Доступна папка приложения, однако здесь пользователь имеет к ней прямой доступ. Файлы могут хранить произ-

вольные атрибуты (пары ключ-значение). Присутствуют аналогичные методы для выполнения операций, перечисленных выше. Стоит отметить, что загрузка файлов на сервер по частям недоступна, а также доступно копирование папок. Файлы имеют одного владельца, но можно делать файлы полностью публичными.

Dropbox[39]. Иерархия папок и файлов, имеющих уникальный идентификатор. Можно адресовать узлы как по пути от корня, так и по идентификаторам. Имена не чувствительны к регистру. Есть папки приложений. Поддерживаются произвольные атрибуты (пары ключ-значение). Доступна аналогичная функциональность, как и у Яндекс.Диска, но доступна загрузка по частям. Есть система контроля доступа к файлам с разной гранулярностью (пользователи и группы).

Amazon S3[40]. Данные хранятся в объектах (objects), расположенных в контейнерах (buckets). Объекты представляют собой содержимое файла и метаданные. Формально контейнер является плоской структурой (т.е. он хранит все объекты без иерархии), но можно организовать иерархию файлов и папок на логическом уровне, используя разделители (/) и общие префиксы для файлов, находящихся в одной директории. Есть система управления доступом к объектам, загрузка по частям, версионирование, произвольные атрибуты, копирование и другие методы, перечисленные выше.

Отдельно стоит отметить, что файловые системы наиболее популярных операционных систем для персональных компьютеров (Windows — NTFS, FAT; дистрибутивы Linux — ext4, btrfs, другие; Mac OS X — APFS, HFS+) тоже имеют свои особенности. Например, файловая система HFS+ по умолчанию настроена так, что имена не чувствительны к регистру. В Windows по умолчанию операции с файловой системой осуществляются в не чувствительном к регистру режиме[41]. В дистрибутивах Linux файловые системы чувствительны к регистру. Данную особенность нужно учитывать при разработке приложений для этих платформ.

2.1.2 Базовые интерфейсы виртуальной файловой системы

Самое близкое к решению целевой задачи из уже существующего (в концептуальном смысле) — это упомянутый в обзоре проект korio[18]. В нем представлен базовый интерфейс виртуальной файловой системы как абстракции, которую реализуют доступные файловые хранилища на конечных платформах. Однако этот интерфейс сильно перегружен: например, там есть методы для наблюдения за событиями файловой системы на определенном файле, но такую функциональность поддерживают не все облачные хранилища (например, Яндекс.Диск не поддерживает), также некоторые абстрактные методы имеют базовые реализации, которые ничего не делают (например, метод `touch`). При попытке поддержать файловое хранилище, может возникнуть ситуация, когда реализация части функциональности не будет возможна ввиду отсутствия такой функциональности у целевого хранилища. В результате, когда пользователь-разработчик будет пытаться использовать определенную функциональность виртуальной файловой системы в общем модуле мультиплатформенного проекта, у него не будет уверенности, что она вообще реализована. Это является существенным недостатком, которого хочется избежать.

Каких дополнительных свойств хочется добиться от конечной библиотеки:

- базовый интерфейс виртуальной файловой системы дает гарантию поддержки минимального набора операций, с помощью которого можно выполнять любые базовые операции над файлами (создание, удаление файлов и папок; запись, чтение, перемещение, копирование файлов) — это позволит быстро обеспечивать базовую поддержку любых хранилищ, которая нужна пользователю;

- поддержка особых возможностей обеспечивается через реализацию дополнительных интерфейсов-расширений виртуальной файловой системы.

Понятно, что для приложения может быть недостаточно лишь базовых операций для работы с файлами. Нужен инструмент для определения контракта

по необходимой функциональности файловых хранилищ, которые будут использоваться на целевых платформах. Эту проблему можно решить на уровне типов языка Kotlin, об этом будет рассказано позже.

Ввиду особенностей различных файловых хранилищ, перечисленных в разделе 2.1.1, предлагается отказаться от адресации узлов по полным путям и ввести на уровне типов отдельно папку и файл как узлы иерархии.

Перейдем к обзору реализованных интерфейсов, которые будут в основе поддержки любого файлового хранилища.

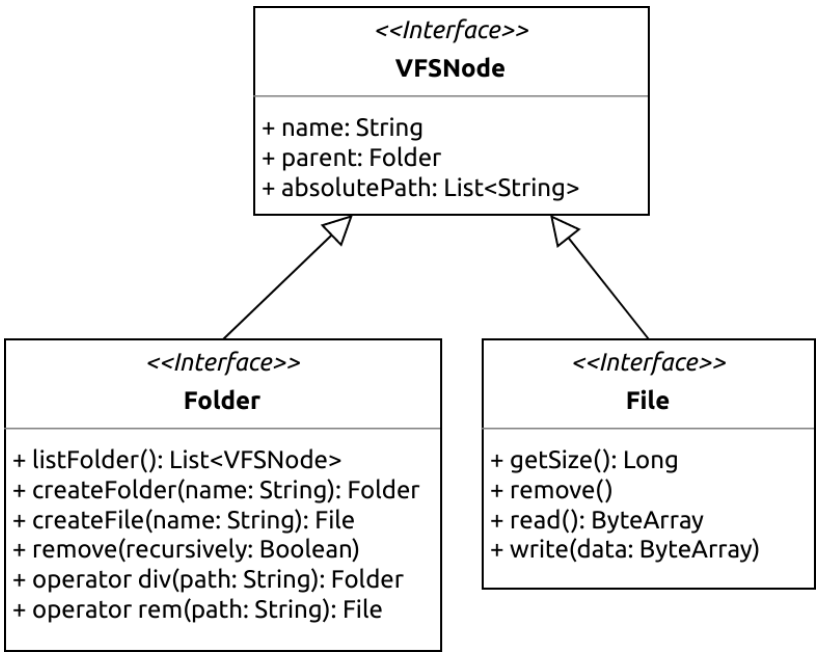

Рис. 1: Интерфейсы узлов виртуальной файловой системы.

На рисунке 1 изображена схема интерфейсов узлов виртуальной файловой системы (VFS).

`VFSNode` — интерфейс, представляющий любой узел VFS. Содержит имя, ссылку на родителя, который всегда является папкой, а также абсолютный путь до этого узла (на самом деле путь храниться в памяти не будет, а будет вычисляемым свойством[42]).

`Folder` — интерфейс, представляющий папку VFS. Содержит методы для получения дочерних узлов; создания папки или файла; удаления узла (флаг `recursively` нужен для определения поведения в случае удаления пустой и непустой папок). Функции-операторы `div` (`/`) и `rem` (`%`) выступают в качестве методов для упрощения навигации по дереву виртуальной фай-

ловой системы. Решение ввести две функции для навигации связано с желанием обеспечить автоматический вывод типа: например, тип выражения `folder / "subfolder"` является `Folder`, а значит можно писать цепочки вида `folder / "subfolder" / "a" / "b" / "c"`, что визуально выглядит как стандартное представление пути в Linux-системах. Получить файл можно, написав выражение вида `folder / "subfolder" % "file.txt"`.

`File` — интерфейс, представляющий файл VFS. Содержит методы для удаления, получения размера в байтах, записи и чтения содержимого целиком как массивов байт.

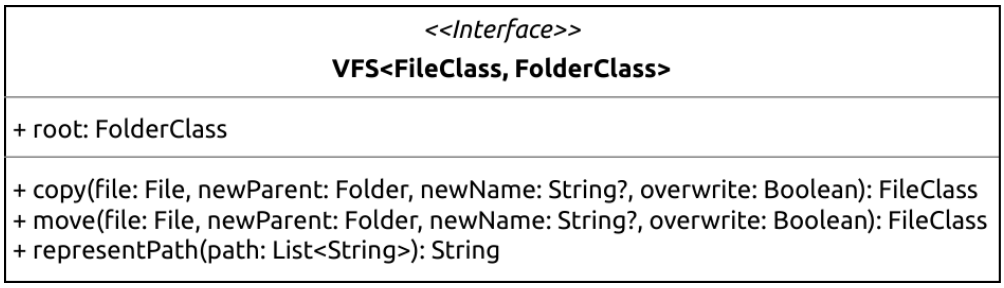

Рис. 2: Интерфейс виртуальной файловой системы.

На рисунке 2 представлен интерфейс непосредственно виртуальной файловой системы. Сразу отметим, что это обобщенный (generic) интерфейс, параметризованный типами файла и папки. Такая запись позволяет включить изначально несвязанные классы, реализующие представленные интерфейсы узлов ВФС, в тип самой ВФС. Это позволит налагать требования на функциональность этих классов на уровне типов. `VFS` имеет свойство `root` — корень файловой системы, а также методы для копирования и перемещения файлов. Метод `representPath` определен для того, чтобы была возможность представлять абсолютные пути в родном для файловой системы виде.

Все перечисленные методы (кроме `representPath`) являются останавливаемыми [43] (suspendable), то есть помечены модификатором `suspend`. Необходимость этого заключается в том, что все операции непосредственно связаны с тем или иным вводом-выводом (от просто системного вызова до отправки HTTP-запроса), а значит существенную часть их исполнения может занимать ожидание. Чтобы эффективнее использовать процессорное время, язык предоставляет механизм кооперативной многозадачности в виде сопрограмм (coroutines), который в данном случае отражен в модификаторе

suspend.

Заметим, что методы copy и move принимают базовые типы File и Folder, а возвращают FileClass — то есть тип файла, который на самом деле предоставляет виртуальная файловая система, реализующая интерфейс. Может возникнуть вопрос, а не должны ли эти методы принимать файлы и папки типов-параметров FileClass и FolderClass, то есть принадлежащих исключительно виртуальной файловой системе, с которой мы работаем. Одна из ключевых концепций использования библиотеки заключается в том, что приложение, которое мы пишем, может предоставлять пользователю выбор из нескольких хранилищ для использования. Это означает, что код в общем модуле должен уметь работать с ВФС разных типов (скажем, VFS<LocalFile, LocalFolder> и VFS<YaDiskFile, YaDiskFolder>). Чтобы это было возможно, нужна ковариантность[44] типовых параметров интерфейса VFS. То есть наш код с логикой приложения может принимать аргумент типа VFS<out File, out Folder>, подтипами которого являются перечисленные типы выше. Однако это не позволяет в методах интерфейса VFS принимать аргументы, которые у́же (в смысле общности типа), чем верхняя граница, то есть File и Folder. Исходя из данного обстоятельства, можно определить семантику методов copy и move следующим образом. Методы принимают в качестве аргументов файлы и папки любых типов-наследников File и Folder, но выполняется проверка времени исполнения соответствия фактического типа папки данной VFS (при несоответствии бросается исключение); если фактический тип файла соотвествует данной VFS, то операции эффективны с точки зрения использования возможностей файлового хранилища, иначе операция может быть неэффективной по ресурсам (например, это может быть чтение содержимого целиком в память и далее запись в целевой файл). Семантика поведения в случае несоответствия файла VFS, вероятно, спорна и может измениться, если библиотека получит развитие.

2.1.3 Расширения

Расширения представляют из себя дополнительные интерфейсы, которые могут реализовывать виртуальные файловые системы.

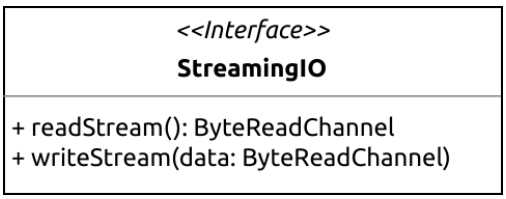

Рис. 3: Интерфейс расширения функциональности файла, обеспечивающий потоковое чтение и запись содержимого.

В качестве демонстрации было реализовано одно расширение, представленное на рисунке 3. `ByteReadChannel` — это интерфейс из библиотеки Ktor (см. раздел 2.3.1), который предоставляет методы потоковой обработки байт. Оба метода помечены модификатором `suspend`.

Интерес представляет то, как работать с этими расширениями в общем коде. Kotlin позволяет задавать несколько верхних границ типовому параметру:

```
suspend fun <F> videoPlayer(fs: VFS<out F, out Folder>)
        where F : File, F : StreamingIO {
  @Suppress("UNCHECKED_CAST")
  val movie = (fs.root / "movies" % "matrix.mp4") as F
  val movieDataStream = movie.readStream()
  playMovie(movieDataStream)
}
```

Как видно из примера, достаточно перечислить требуемую функциональность в блоке `where`. Так как методы для навигации возвращают лишь базовые типы `File` и `Folder`, приходится делать явное преобразование типов. Но заметим, что в данном случае оно безопасно, поскольку файл `movie` получен путем навигации по дереву `fs`, а в типе `fs` заявлен тип файла `F`. Была осуществлена попытка реализовать навигацию, позволяющую обойтись без приведения типа, но она окончилась неудачей.

Из неудобств данной записи можно отметить то, что нельзя завести синоним типа (`typealias`), сочетающий в себе несколько верхних границ. Kotlin позволяет заводить синоним типа вида

```
typealias Storage = VFS<out File, out Folder>,
```

но на текущий момент язык не позволяет записи вида

```
typealias Storage = VFS<out (File & StreamingIO), out Folder>,
```

однако авторы языка рассматривают[45] добавление такой возможности.

Среди базовых расширений, которые можно было бы дополнительно реализовать, можно назвать следующие: поддержка меток времени создания, доступа и модификации файла, прав доступа на чтение и запись; поддержка атомарных операций; возможность управлять произвольными атрибутами (пары ключ-значение).

2.1.4 Инварианты и гарантии

- Свойство `parent` интерфейса `VFSNode` указывает на родительскую папку для данного узла. Для корневой папки это свойство ссылается на сам корень. Для удобства разработки для интерфейса `Folder` добавлено булево свойство-расширение `isRoot`.
- Удаление папки без явно установленного флага `recursively` возможно только в случае, если папка пуста. Иначе бросается исключение, помеченное `VFSFolderNotEmptyException`.
- Библиотека не накладывает никаких ограничений на имена файлов и папок. Пользователь библиотеки должен самостоятельно проверить совместимость используемых файловых хранилищ с логикой приложения и при необходимости ограничить конечного пользователя в именовании файлов и папок.
- Методы для навигации проверяют существование файла или папки: если узла не существует, бросается исключение, помеченное `VFSFileNotFoundException` или `VFSFolderNotFoundException`.
- Классы файлов и папок переопределяют методы `equals` и `hashcode` в соответствии с семантикой ВФС.
- Методы `copy` и `move` принимают любые файлы и папки, однако фактический тип папки должен соответствовать виртуальной файловой системе.

Гарантируется эффективная реализация этих методов, только если фактический тип файла соотвествует ВФС. В противном случае не дается гарантий на эффективность (возможная реализация — чтение файла целиком в память и его запись в целевой файл).

Заметим, что для сохранения гарантии достаточно обеспечить, чтобы операции над узлами ВФС не покидали ее пределов.

- Все методы ВФС или её узлов могут бросить исключения только типов-наследников `VFSException`. Вместе с сообщением об ошибке передается также исключение-причина в поле `cause`, если оно было брошено внутри реализации ВФС. Для возможности более гранулярной ловли исключений добавлены интерфейсы-маркеры: `VFSNodeNotFoundException` с двумя наследниками для файла и папки и аналогичные маркеры `VFSNodeExistsException`.

2.1.5 Преимущества и недостатки выбранного подхода

Чего удалось добиться представленной архитектурой:

- удобная и понятная навигация с выводом типов;
- ВФС предоставляет только ту функциональность, которая на самом деле доступна;
- есть возможность расширения функциональности ВФС и наложения требований на их наличие на уровне типов;
- базовые интерфейсы просты и лаконичны, их сравнительно легко реализовать для нового файлового хранилища;
- платформоспецифичный код приложений получается очень ёмким (будет показано в примерах позже).

Из недостатков можно отметить следующие:

- при работе с расширениями приходится использовать обобщенные функции, а также каждый раз указывать типовые ограничения;

- также при работе с расширениями приходится явно приводить типы конечных файлов при навигации, хоть это и безопасно;
- неочевидная семантика методов `copy` и `move`, если использовать их на файлах, не относящихся к ВФС.

2.2. Файловые хранилища, поддержанные библиотекой

В этом разделе рассказывается о поддержанных хранилищах. Технические детали реализации описаны в разделе 3.

В таблице 1 содержится сводка о поддержке возможностей каждой реализованной ВФС.

	Платформа			Расширения
Название	JVM	Android	JS (browser)	`StreamingIO`
`SystemFS`	да	да		да
`GoogleDriveFS`	да	да	да	да
`SqliteFS`		да		

Таблица 1: Сводная таблица поддерживаемых платформ и расширений реализованными ВФС.

2.2.1 `SystemFS`

`SystemFS` — это ВФС, которая предоставляет доступ к локальной файловой системе посредством функциональности `java.nio.Path`, доступна на платформах JVM и Android.

На рисунке 4 изображена схема наследования классов данной ВФС. Узел `SystemFSNode` содержит свойство `nioPath`, представляющего путь в локальной файловой системе до этого узла. Реализация методов узлов по сути состоит из вызова соответствующего метода `nioPath` и обработки ошибок. `java.nio.Path` предоставляет функциональность для потокового чтения и записи, так что реализация расширения `StreamingIO` тоже получилась очень простой.

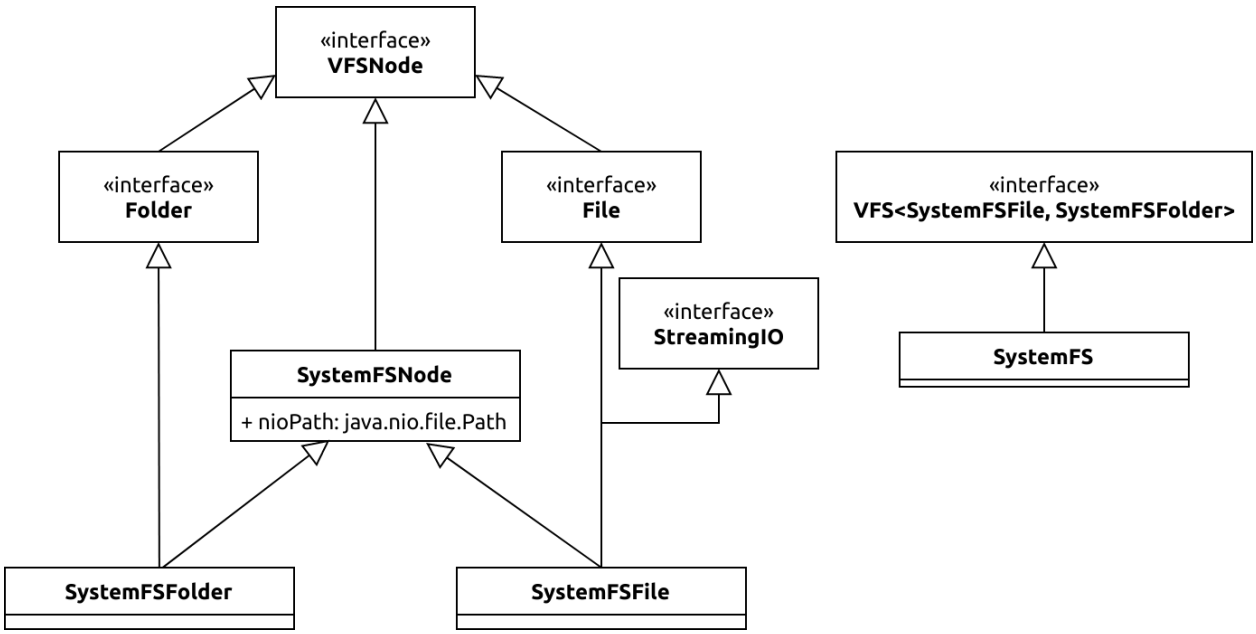

Рис. 4: Диаграмма классов `SystemFS`.

Конструктор класса `SystemFS` принимает путь до папки, которая будет корнем ВФС, по умолчанию установлено значение пути до текущей рабочей директории. Этот механизм позволяет ограничивать операции над ВФС пределами одной директории настоящей файловой системы.

2.2.2 `GoogleDriveFS`

`GoogleDriveFS` — это ВФС, которая предоставляет доступ к содержимому Google Drive пользователя. Коммуникация с сервисом осуществляется через REST HTTP API с помощью библиотеки Ktor. Доступна на всех платформах.

Для использования необходимо зарегистрировать приложение в консоли Google Cloud[46], настроить его, добавив в список API доступ к Google Drive, и получить учетные данные (идентификатор приложения и секретный ключ).

На рисунке 5 изображена архитектура поддержки этого файлового хранилища. Видим, что в данном случае код имеет платформоспецифичные компоненты (`GoogleAuthorizationRequester` для платформ JVM, Android и JS (browser)). В общем модуле расположена вся основная логика для обеспечения работы ВФС: это сами классы ВФС и ее узлов (с аналогичной схе-

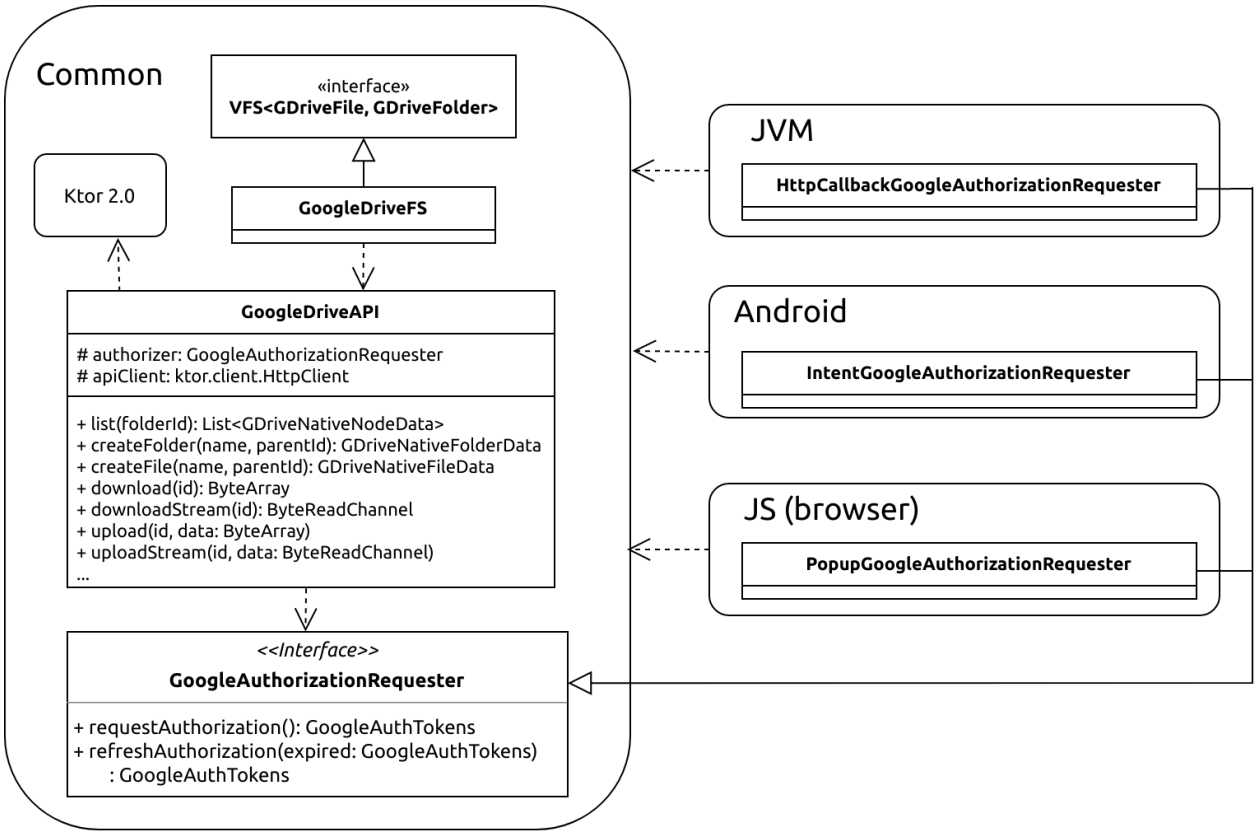

Рис. 5: Архитектура реализации поддержки файлового хранилища Google Drive.

мой наследования как на рисунке 4), которым для работы нужен экземпляр `GoogleDriveAPI` — это по сути набор методов для совершения HTTP-запросов в сервис Google Drive. Для совершения HTTP-запросов используется мультиплатформенная библиотека Ktor. Чтобы совершать HTTP-запросы от имени пользователя, необходимо получить ключ доступа (access token) к пространству пользователя, для этого необходимо пройти процедуру авторизации по протоколу OAuth 2.0[47]. Для этого нужно каким-то образом отправить пользователя в сервис авторизации Google, где он подтвердит выдачу авторизации нашему приложению. На разных платформах это делается по-разному.

- Если наше приложение работает в веб-браузере, то можно воспользоваться средствами JavaScript-библиотеки Google Platform Library, которая сама проводит пользователя через весь процесс, открывая дополнительную вкладку в браузере, и возвращает готовые ключи в JavaScript-объекте.

- Если наше приложение работает на платформе JVM (на персональном компьютере), то один из вариантов провести авторизацию работает следующим образом. Приложение запускает HTTP-сервер, далее открывает страницу в браузере со специальной ссылкой на сервис авторизации Google, куда добавлена информация об адресе созданного HTTP-сервера. Когда пользователь подтверждает авторизацию доступа приложению, страница отправляет HTTP-запрос с кодом авторизации по указанному адресу, нам остается его получить и обменять на ключи доступа.

- Если наше приложение работает на устройстве под управлением Android, то можно воспользоваться библиотекой `play-services-auth`, в которой есть `SignInIntent` (если огрубить, то это всплывающий экран, на котором пользователю предлагается выбрать аккаунт для авторизации в один клик), из результата которого можно достать код авторизации.

Получается следующая схема работы: `GoogleDriveAPI` принимает экземпляр интерфейса `GoogleAuthorizationRequester`, позволяющего ему при необходимости получать код доступа или обновлять его. На каждой платформе есть своя реализация этого интерфейса.

При инициализации `GoogleDriveFS` можно указывать пространство (scope), в котором приложение хочет работать (все файлы пользователя или приватная папка приложения).

Благодаря мультиплатформенности библиотеки Ktor, весь код для совершения запросов и управления файлами на логическом уровне полностью переиспользуется (около 650 строк кода), платформоспецифичные компоненты сравнительно небольшие и занимают для платформ JS (browser) и Android около 50 строк кода и около 100 для JVM.

2.2.3 `SqliteFS`

`SqliteFS` — это ВФС, которая эмулирует файловую систему внутри базы данных SQLite. Доступна на платформе Android. Поскольку возможность создавать и использовать базы данных SQLite является встроенной

возможностью платформы, а также с целью демонстрации эмуляции файловой системы с помощью иных способов хранения на базе библиотеки, было решено реализовать поддержку такой ВФС.

Расширение StreamingIO не реализовано, поскольку нет возможности читать и записывать данные в базу данных SQLite в потоковом режиме.

2.3. Используемые зависимости

Помимо двух официальных языковых библиотек для работы с (де-)сериализацией JSON объектов и работы с корутинами, библиотека `multifs` имеет следующие зависимости.

2.3.1 Ktor

Ktor[32] — это библиотека для построения веб-приложений и HTTP-сервисов, которая официально поддерживается компанией JetBrains, создавшей язык Kotlin. В нашей задаче эта библиотека используется как средство, обеспечивающее мультиплатформенную работу с HTTP-соединениями, а также используются примитивы этой библиотеки для потоковой обработки данных: `ByteChannel`, `ByteReadChannel`.

2.3.2 Зависимости на платформе Android

На платформе Android для работы ВФС `GoogleDriveFS` необходимо подключение двух дополнительных зависимостей:

- `activity-ktx`[48] — предоставляет расширения существующих библиотек платформы Android для языка Kotlin.
- `play-services-auth`[49] — содержит функциональность для получения авторизации доступа к Google Drive (экран `SignInIntent`).

3. Детали реализации

3.1. Структура проекта

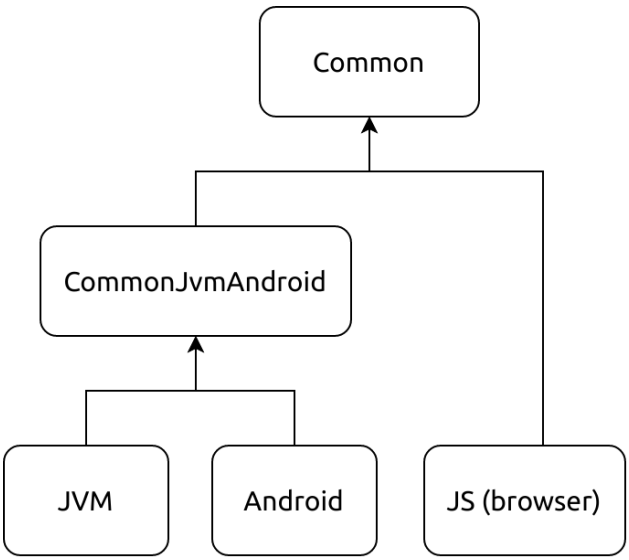

Рис. 6: Структура модулей проекта `multifs`.

Библиотека `multifs`[34] разделена на пять модулей, представленных на рисунке 6. Модуль Common содержит определения базовых интерфейсов библиотеки, а также основную часть логики для поддержки Google Drive (`GoogleDriveAPI` и `GoogleDriveFS`). Модуль CommonJvmAndroid нужен для реализации поддержки `SystemFS`, так как необходимая для этого функциональность доступна на обеих платформах JVM и Android. Модули JVM и JS (browser) содержат реализации интерфейса `GoogleAuthorizationRequester`, обеспечивающих получение кода доступа к Google Drive пользователя. В модуле Android, помимо реализации этого интерфейса, содержится также реализация `SqliteFS`.

3.2. `GoogleDriveFS`

3.2.1 `GoogleDriveAPI`

`GoogleDriveAPI` использует HTTP-клиент библиотеки Ktor с подключаемым модулем `Auth`, который самостоятельно обеспечивает добавление HTTP-заголовка `Bearer` с ключом доступа и при необходимости вызывает методы для получения или обновления этого ключа.

Так как большинство реализованных методов по своей механике не сильно друг от друга отличаются, здесь будет дан обзор наиболее интересных из них.

Метод list. В качестве аргумента принимает идентификатор папки, список детей которой необходимо получить. Совершает GET-запросы по адресу `https://www.googleapis.com/drive/v3/files` с дополнительным параметром для получения детей именно указанной папки, в теле ответа возвращается список файлов (возможно, частичный) с метаинформацией в виде JSON-документа. Если в теле ответа указано поле `nextPageToken`, то полученный список не является полным и выполняются дополнительные запросы, чтобы получить следующие части ответа. Тело ответа десериализуется в JSON-объект с помощью библиотеки `kotlinx.serialization.json`.

Метод download. В качестве аргумента принимает идентификатор файла, который необходимо скачать. Выполняет GET-запрос по адресу `https://www.googleapis.com/drive/v3/files/$id` с параметром `alt=media`, который обозначает опцию получения именно содержимого файла. Полностью считывается тело ответа, которое и является содержимым файла.

Метод downloadStream. Этот метод, в отличие от предыдущего, предназначен для получения содержимого файла в потоковом режиме. Для этого создается экземпляр `ByteChannel`, который возвращается из функции. В качестве побочного эффекта в области видимости сопрограмм (coroutine scope) HTTP-клиента запускается сопрограмма, которая выполняет запрос и копирует (потоково) тело ответа в ранее созданный канал. В случае исключения, канал закрывается с информацией о брошенном исключении. При завершении получения ответа канал закрывается.

Метод upload. В качестве аргументов принимает идентификатор файла и его содержимое в виде массива байт. Выполняется PATCH-запрос по адресу

`https://www.googleapis.com/upload/drive/v3/files/$id` с параметром `uploadType=media`, в теле которого записано полное содержимое файла.

Метод uploadStream. В качестве аргументов принимает идентификатор файла и `ByteReadChannel`, из которого можно вычитать содержимое файла. Сначала выполняется аналогичный PATCH-запрос для получения адреса возобновляемой загрузки данных (в данном случае значение параметра `uploadType=resumable`), по которму можно загружать содержимое файла частями. Сервис Google Drive налагает ограничение на размер загружаемых частей: размеры всех частей, кроме завершающей, должны быть кратны 256 КиБ. Алгоритм загрузки следующий. Вычитываются 256 КиБ данных из входного канала (либо меньше, если канал закрылся). Полученный кусок (chunk) загружается по полученному адресу с помощью метода PUT и заголовка `Content-Range`, который содержит местоположение этих данных в файле. В случае ошибок выполняется повторный запрос с промежутком в 250 мс, в случае успеха счетчик повторных запросов сбрасывается, а при превышении порогового значения (4 попытки) бросается исключение. Данная процедура повторяется, пока не будут загружены все данные из входного канала.

Реализованный алгоритм нельзя назвать оптимальным, поскольку, во-первых, можно вычитывать следующий кусок данных, пока предыдущий загружается, во-вторых, можно масштабировать размер отправляемых данных для лучшей утилизации пропускной способности, но для этого нужно реализовать соответствующий механизм, который будет подстраиваться под качество интернет-соединения.

3.2.2 `GoogleAuthorizationRequester` и его реализации

Как обсуждалось в разделе 2.2.2, процесс получения авторизации к Google Drive пользователя реализуется по-разному на каждой из платформ.

JVM. На платформе JVM доступен класс `HttpCallbackGoogleAuthorizationRequester`. Метод `requestAuthorization` реализован следующим образом. Создается перемен-

ная tokensFuture типа CompletableDeferred<GoogleAuthTokens>, в которую будут записаны ключи доступа. Запускается HTTP-сервер на случайном порте, ожидающий HTTP-запрос кодом авторизации. Далее с помощью системных средств открывается страница в веб-браузере со специально сформированным адресом сервиса авторизации Google, где пользователю предлагается выбрать свой аккаунт и подтвердить выдачу авторизации. После получения разрешения, веб-страница посылает запрос с кодом авторизации на адрес запущенного HTTP-сервера. Получив код авторизации, выполняется HTTP-запрос для его обмена на ключ доступа, после чего полученные ключи записываются в переменную tokensFuture. После того как пользователь направлен в веб-браузер, выполняется ожидание результата на переменной tokensFuture. После этого запущенный HTTP-сервер останавливается и результат в виде полученных ключей возвращается.

Android. На платформе Android доступен класс IntentGoogleAuthorizationRequester. Для получения кода авторизации используются средства библиотеки play-services-auth. Данный класс в конструкторе принимает, кроме учетных данных приложения и пространства (scope) Google Drive, ещё контекст Android-приложения и лямбда-функцию типа suspend (Intent) -> ActivityResult. Задача этой лямбда-функции — запустить переданный Intent (намерение) и вернуть после завершения его результат. Ввиду особенностей платформы Android, данную процедуру нельзя выполнить, не затрагивая код самого приложения (во всяком случае автору такие способы не известны). Однако, как показано в примерах, при использовании современной библиотеки Compose, можно реализовать эту функцию, не затрагивая существующий код.

В данном классе при инициализации создается объект типа GoogleSignInClient, который содержит Intent получения авторизации. При вызове метода requestAuthorization происходит запуск данного намерения через упомянутую лямбда-функцию. Со стороны пользователя это выглядит как всплывающее окно со списком учетных записей. После клика по одному из них окно закрывается, а результат передается для обработки в наш метод. Далее из результата достается код авторизации, который меняется на ключ

доступа.

Полученные ключи доступа, ввиду требований безопасности, имеют срок годности порядка одного часа, из-за чего их необходимо периодически обновлять. Для этого выполняется HTTP-запрос по адресу `https://oauth2.googleapis.com/token`, параметризованный ключом обновления (refresh token). Поскольку данная процедура общая для реализаций JVM и Android, она вынесена в Companion Object интерфейса `GoogleAuthorizationRequester`. При неуспешном обновлении ключа доступа, вызывается `requestAuthorization` для получения нового ключа.

JS (browser). Для браузерных приложений доступен класс `PopupGoogleAuthorizationRequester`. Для его работы отдельно требуется, чтобы html-страница приложения загружала Google Platform Library[1], доступную по адресу `https://apis.google.com/js/platform.js`, а также инициализировала её модуль `gapi.auth2` с помощью идентификатора используемого приложения.

Реализация метода `requestAuthorization` представляет из себя вызов метода `signIn` на объекте, который порождается упомянутой библиотекой. Этот метод возвращает `Promise`-объект[50], результатом которого является объект с информацией об учетной записи, содержащей также и код доступа. При вызове метода `signIn` открывается всплывающее окно с предложением выбрать учетную запись для доступа к Google Drive. После выбора учетной записи, окно закрывается, а результат завершает созданный `Promise`.

Обновление ключа доступа происходит аналогично через вызов метода `reloadAuthResponse` на полученном ранее объекте учетной записи пользователя.

3.3. `SqliteFS`

Для того, чтобы сэмулировать иерархическую структуру ВФС, в базе данных SQLite создаются две таблицы: папки и файлы. Таблица папок имеет

[1] к сожалению, слишком поздно выяснилось, что эта библиотека объявлена устаревшей и перестанет работать в конце марта 2023 года, однако заменить её в будущем на новую библиотеку Sign In With Google не должно составить большого труда

поля идентификатора (первичный ключ с автоинкрементом), имени и ссылки на родительскую папку. Таблица файлов устроена аналогично, но дополнительно имеет поле содержимого файла типа `blob`.

Эти таблицы создаются с ограничением уникальности пары `(name, parent)`, что обеспечивает существование не более одной папки или файла с выбранным именем внутри родительской папки. Также добавлено ограничение на ненулевую длину имени узла, которое не распространяется на корневую папку. Корневая папка добавляется отдельно после создания таблицы и имеет идентификатор 0 и пустое имя.

Описание таблиц и команды их создания вынесены в отдельный объект в файле `SQLContract.kt`.

Конструктор `SqliteFS` принимает в качестве единственного аргумента экземпляр класса `SqliteFSDatabaseHelper`, являющегося наследником класса `SQLiteOpenHelper`, в котором переопределены методы для первоначального создания таблиц в базе данных. Данный класс наследует свойства `writableDatabase` и `readableDatabase`, которые уже представляют экземпляры класса `SQLiteDatabase`, имеющего методы для совершения самих SQL-запросов.

Сами запросы не представляют большого интереса, для примера прилагается самый сложный запрос из реализации метода `copy`:

```
db.execSQL(
  """
    INSERT INTO ${Files.TABLE_NAME}
      (${Files.COLUMN_NAME}, ${Files.COLUMN_PARENT}, ${Files.COLUMN_DATA})
    SELECT
      ? ${Files.COLUMN_NAME},
      ${newParent.id} ${Files.COLUMN_PARENT},
      ${Files.COLUMN_DATA}
    FROM ${Files.TABLE_NAME}
    WHERE ${Files.COLUMN_ID} = ${file.id};
  """.trimIndent(),
  arrayOf(targetName)
)
```

3.4. Тестирование

Автоматические тесты реализованы только для ВФС `SystemFS`, проверяются в основном лишь базовые сценарии. Всего написано 7 тестов. Встроенное в среду разработки IntelliJ IDEA CE средство подсчета покрытия кода показывает 53% покрытие по методам и 34% покрытие по строкам в модуле `dev.salavatov.multifs.systemfs`.

Корректность работы реализованной библиотеки проверялась в ручном режиме через реализованные на ее базе приложения, о которых рассказывается в разделе 4.

3.5. Прочее

Обобщенные реализации методов `move` и `copy`. Как было упомянуто в разделе 2.1.2, семантика методов `move` и `copy` предполагает, что в случае несоответствия типа переданного в метод файла типу файла самой ВФС, операция выполняется, однако, возможно, с чтением содержимого в память полностью. Данное поведение является общим для всех реализаций ВФС, поэтому оно выделено в отдельные встраиваемые (inline) обобщенные методы `genericCopy` и `genericMove` с параметрами вещественного (reified) типа[51] `FileClass` и `FolderClass`. Reified-параметры нужны, поскольку в JVM не сохраняется информация о типовых параметрах.

4. Примеры использования библиотеки

4.1. Сборка проекта

Сборка проекта осуществляется с помощью системы автоматической сборки Gradle[52] версии 7.1, используемая версия JDK — OpenJDK 11. Сценарий сборки написан в файле `build.gradle.kts` в корне проекта. Используемая версия языка Kotlin — 1.6.20.

Осуществить сборку и опубликовать артефакты в локальный maven-репозиторий можно с помощью команды `./gradlew publishToMavenLocal`.

4.2. Multieditor

Multieditor[53] — это мультиплатформенный текстовый редактор, доступный на платформах JVM, Android и Web и предоставляющий возможность использовать все три поддержанных ВФС: `SystemFS`, `GoogleDriveFS`, `SQLiteFS`.

4.2.1 Подключение библиотеки

Подключение библиотеки `multifs` осуществляется через сценарий автоматической сборки в файле `build.gradle.kts`. Поскольку на текущий момент библиотека ещё не опубликована в популярных maven-репозиториях, используется локальный maven-репозиторий. В блок `dependencies` общего модуля проекта добавлена зависимость

```
implementation("dev.salavatov:multifs:${Versions.multifs}").
```

В модули для платформ jvm, android и js добавлена соотвествующая зависимость на платформоспецифичный артефакт этой библиотеки: `multifs-jvm`, `multifs-android` или `mutlifs-js`.

4.2.2 Реализация бизнес-логики в общем модуле приложения

Вся основная логика приложения написана в файле `AppState.kt` в общем модуле. Класс `AppState` содержит поля, представляющие состояние области ввода текста (информация о текущем файле, содержимое, происходит

ли сохранение данных в текущий момент), дерева навигации (список доступных для подключения хранилищ, состояние деревьев подключенных виртуальных файловых систем), а также последней ошибки.

Этот класс также содержит собственную область видимости сопрограмм, диспетчеризацию которой осуществляет диспетчер сопрограмм, получаемый из функции `MultifsDispatcher()` — это функция, объявленная с помощью `expect/actual`-механизма. На платформах JVM и Android эта функция возвращает `Dispatchers.IO`, на платформе JS (browser) — `Dispatchers.Unconfined`.

Определен метод `launchSafe`, который запускает произвольный `suspend`-блок кода в упомянутой области видимости сопрограмм, при этом все исключения этого блока ловятся и сохраняются в поле `errorState` для возможности демонстрации ошибки пользователю.

С помощью этого метода уже реализуется непосредственно бизнес-логика приложения (10 функций), несколько из них для примера: **(TODO: надо ли выделить код в нумерованный листинг?)**

```
fun launchFileOpen(targetFile: File) =
  launchSafe {
    val contentBytes = targetFile.read()
    if (!editor.saving) {
      editor.content = contentBytes.decodeToString()
      editor.file = targetFile
    }
  }

fun launchInitializeStorage(namedStorageFactory: NamedStorageFactory) =
  launchSafe {
    val fs = namedStorageFactory.init()
    val name = namedStorageFactory.name
    navigation.availableStoragesState.remove(namedStorageFactory)
    navigation.configuredStoragesState.add(
      FileTree(
        fs,
        name,
        FolderNode(fs.root, null, mutableStateListOf())
     )
```

```
    )
  }
```

В последней приведенной функции конструктор `FolderNode` имеет параметры: `Folder` из `multifs`, ссылка на родительский `FolderNode`, состояние списка дочерних узлов (оно обновляется при раскрытии/скрытии подробной информации этой папки).

Реализации графических интерфейсов для изменения состояния приложения вызывают только методы класса `AppState`.

4.2.3 Определение списка доступных хранилищ на каждой из целевых платформ

Как можно было заметить в коде выше, бизнес-логика приложения работает со списком доступных хранилищ `navigation.availableStoragesState`. Для работы с хранилищами в общем коде создан вспомогательный класс и синоним типа:

```
typealias Storage = VFS<out File, out Folder>
class NamedStorageFactory(val name: String, val init: suspend () -> Storage)
```

Получение списка доступных хранилищ на каждой платформе вынесено в функцию или класс:

```
// JVM
fun makeStorageList(): List<NamedStorageFactory> {
  val systemfs = NamedStorageFactory("Local device") { SystemFS() }
  val gdfs = NamedStorageFactory("Google Drive") {
    val googleAuth = CacheGoogleAuthorizationRequester(
      HttpCallbackGoogleAuthorizationRequester(
        GoogleAppCredentials(CLIENT_ID, SECRET),
        GoogleDriveAPI.Companion.DriveScope.General
      )
    )
    val gapi = GoogleDriveAPI(googleAuth)
    GoogleDriveFS(gapi).also {
      it.root.listFolder() // trigger auth
    }
```

```
  }
  return listOf(systemfs, gdfs)
}
```

Вызов `listFolder` на корне ВФС Google Drive нужно, чтобы при инициализации хранилища сразу произошел запрос авторизации.

```
// JS
fun makeStorageList(): List<NamedStorageFactory> {
  val gdfs = NamedStorageFactory("Google.Drive") {
    val googleAuth =
      PopupGoogleAuthorizationRequester(
        GoogleDriveAPI.Companion.DriveScope.General
      )
    val gapi = GoogleDriveAPI(googleAuth)
    GoogleDriveFS(gapi).also {
      it.root.listFolder() // trigger auth
    }
  }
  return listOf(gdfs)
}

// Android
class Storages(
  private val googleAuth: IntentGoogleAuthorizationRequester,
  private val sqliteFSDbHelper: SqliteFSDatabaseHelper,
  private val systemFSRoot: Path
) {
  private val sqlite = NamedStorageFactory("SQLite") {
      SqliteFS(sqliteFSDbHelper)
  }

  private val googleDrive = NamedStorageFactory("Google Drive") {
    val gapi = GoogleDriveAPI(googleAuth)
    GoogleDriveFS(gapi).also {
        it.root.listFolder() // trigger auth
    }
  }

  private val systemFS = NamedStorageFactory("Local") {
```

```
    SystemFS(systemFSRoot)
  }

  fun asList(): List<NamedStorageFactory> {
    return listOf(googleDrive, sqlite, systemFS)
  }
}
```

В случае Android это реализовано в виде класса, который в конструкторе принимает необходимые для инициализации ВФС компоненты. Первый из них — это экземпляр `IntentGoogleAuthorizationRequester`. Как было упомянуто в разделе 3.2.2, для создания экземпляра этого класса необходимо определить лямбда-функцию, которая запускает переданный в нее `Intent` и возвращает результат этого намерения. Клиент данного приложения для платформ JVM и Android написан с помощью мультиплатформенной библиотеки Compose[54]. С помощью средств этой библиотеки возможно реализовать создание экземпляра класса `IntentGoogleAuthorizationRequester`, не изменяя существующий код:

```
@Composable
fun makeGoogleAuthorizationRequester(
  googleAppCredentials: GoogleAppCredentials,
  scope: GoogleDriveAPI.Companion.DriveScope
): IntentGoogleAuthorizationRequester {
  var resultFuture = remember { CompletableDeferred<ActivityResult>() }
  val startForResult =
    rememberLauncherForActivityResult(
        ActivityResultContracts.StartActivityForResult()
    ) {
        resultFuture.complete(it)
    }
  return IntentGoogleAuthorizationRequester(
    googleAppCredentials,
    scope,
    LocalContext.current,
  ) { intent ->
    resultFuture = CompletableDeferred<ActivityResult>()
    startForResult.launch(intent)
```

```
    resultFuture.await()
  }
}
```

Как видно из листинга, используется аннотация `@Composable`, а также функция `rememberLauncherForActivityResult`.

Можно обойтись и стандартными средствами библиотеки Android, но возникнет необходимость переопределять методы класса `Activity` для обработки результатов запущенного намерения (`onActivityResult`), либо зарегистрировать `ActivityResultLauncher` в существующей `Activity` (`registerForActivityResult`).

4.2.4 Реализация клиентских приложений на разных платформах

Для платформ JVM и Android код клиента с графическим интерфейсом пользователя написан в общем для них модуле с помощью библиотеки Compose[54]. Для платформы JS (browser) клиент реализован аналогично с помощью библиотеки Compose for Web.

При запуске приложения пользователю в меню навигации доступны несколько ВФС, которые можно подключить. При попытке подключить Google Drive открывается вкладка в браузере, где пользователю предлагается выбрать учетную запись Google (рисунок 8). После выбора аккаунта можно пользоваться подключенной ВФС как обычной файловой системой (рисунок 9). Пользователю доступен для использования весь функционал библиотеки (создание и удаление папок и файлов, просмотр файлов, их сохранение, перемещение, копирование).

Тот же самый клиент доступен на платформе Android (рисунки 10, 11, 12). Для платформы JS (browser) клиент выглядит слегка иначе, но предоставляет в точности те же самые возможности (рисунок 13).

Для создания диалоговых окон библиотека Compose предоставляет `AlertDialog`, однако на текущий момент при его использовании возникают различные ошибки[55, 56], поэтому используется своя обертка через `expect/actual`-механизм: на платформе JVM используется `Dialog` для создания нового системного окна с переопределенными декорациями, на Android

— тоже Dialog, но ввиду специфики код несколько отличается. Для JS (browser) в код страницы добавляется глобальный элемент модального окна, который управляется через модификацию содержимого и CSS-стили.

4.3. gdrive-cli

gdrive-cli[57] — это приложение, предоставляющее возможность работать с хранилищем Google Drive через интерфейс командной строки. Позволяет в том числе загружать и скачивать файлы в потоковом режиме. Писалось с целью проверки работоспособности данной функциональности.

4.3.1 Написание логики приложения с учетом нескольких типовых ограничений на функциональность VFS

Логика приложения написана в функции runCli, имеющей следующую сигнатуру:

```
suspend fun <T> runCli(fs: VFS<out T, out Folder>)
        where T : File, T : StreamingIO {
  ...
}
```

Как видно по типу ВФС, данная функция может принимать не только GoogleDriveFS, но также любую другую виртуальную файловую систему, класс файла которой реализует расширение StreamingIO.

Для примера разберем как реализована команда скачивания файла:

```
when (command) {
  // ...
  is Download -> {
    val node = list[command.index - 1] // Узел, который хотим скачать
    // Проверяем, что мы хотим скачать файл, а не папку. Поскольку тип
    // node - это VFSNode, необходимо приведение типов также к StreamingIO.
    // Можно обойтись без явной проверки, поскольку файл класса точно
    // реализует это расширение. В данном случае с помощью этой проверки
    // срабатывает умное приведение типов Kotlin.
    if (node !is File || node !is StreamingIO)
      throw Exception("cannot download a directory")
```

```
    // Получаем канал, из которого можно читать содержимое файла
    val dlStream = node.readStream()
    // Определяем в какой файл записывать данные
    val destPath = Paths.get(command.destPath)
    if (destPath.exists()) {
      destPath.deleteExisting()
    }
    val destFile = destPath.toFile()
    while (!dlStream.isClosedForRead) {
      // Пока в канале есть данные, считываем их по частям и дополняем
      // файл, попутно логируем в консоль количество записанных байт
      dlStream.read { buffer ->
        print("\rbytes: ${dlStream.totalBytesRead}")
        destFile.appendBytes(buffer.moveToByteArray())
      }
    }
    // Пишем в консоль размер скачанного файла
    println("\rtotal bytes downloaded: ${dlStream.totalBytesRead}")
  }
  // ...
}
```

При использовании данной команды можно наблюдать, что в консоли в течение всего процесса скачивания обновляется сообщение о размере записанных байт.

Схожим образом написана команда загрузки данных в Google Drive.

С помощью данной утилиты проверена функциональность потокового чтения и записи файлов для ВФС `GoogleDriveFS`: были несколько раз успешно загружены и скачаны файлы размером в 50 МиБ.

Заключение

Проведено исследование существующих решений и выявлены их достоинства и недостатки.

Основным результатом данной работы является реализованная мультиплатформенная Kotlin-библиотека `multifs`, позволяющая описывать логику работы с виртуальной файловой системой в общем коде. Данная библиотека предоставляет поддержку трех файловых хранилищ, среди которых одно облачное, доступное на платформах JVM, Android и JS (browser).

На базе созданной библиотеки реализовано два приложения, одно из которых мультиплатформенное с клиентами на трёх перечисленных платформах и поддерживающее все три файловых хранилища. Данное приложение демонстрирует возможности и процесс использования `multifs` с точки зрения разработчика.

Ссылки

[1] *File IO with Kotlin multiplatform - Stack Overflow*. URL: https://stackoverflow.com/q/68191209. (дата обращения 04.05.2022).

[2] *Read/Write file in kotlin native - IOS side - Stack Overflow*. URL: https://stackoverflow.com/q/61078285. (дата обращения 04.05.2022).

[3] *Multiplatform File I/O - r/Kotlin (reddit)*. URL: https://www.reddit.com/r/Kotlin/comments/g0hjsw/multiplatform_file_io/. (дата обращения 04.05.2022).

[4] *Kotlin/native: library for file io? - r/Kotlin (reddit)*. URL: https://www.reddit.com/r/Kotlin/comments/s3wzmt/kotlinnative_library_for_file_io/. (дата обращения 04.05.2022).

[5] *Looking for a library or an alternative for basic multiplatform File I/O - Kotlin Discussions*. URL: https://discuss.kotlinlang.org/t/looking-for-a-library-or-an-alternative-for-basic-multiplatform-file-i-o/21472. (дата обращения 04.05.2022).

[6] *Kotlin Multiplatform*. URL: https://kotlinlang.org/docs/multiplatform.html. (дата обращения 13.05.2022).

[7] *Kotlin for JavaScript*. URL: https://kotlinlang.org/docs/js-overview.html. (дата обращения 13.05.2022).

[8] *Differences between Node.js and the Browser*. URL: https://nodejs.dev/learn/differences-between-nodejs-and-the-browser. (дата обращения 13.05.2022).

[9] *Virtual file system*. URL: https://en.wikipedia.org/wiki/Virtual_file_system. (дата обращения 11.05.2022).

[10] *StackOverflow*. URL: https://stackoverflow.com. (дата обращения 04.05.2022).

[11] *GitHub*. URL: https://github.com. (дата обращения 04.05.2022).

[12] *Kotlin Multiplatform Libraries*. URL: https://github.com/AAkira/Kotlin-Multiplatform-Libraries. (дата обращения 04.05.2022).

[13] *kile*. URL: https://github.com/realad/kile. (дата обращения 04.05.2022).

[14] *File Transfer Protocol - Wikipedia*. URL: https://en.wikipedia.org/wiki/File_Transfer_Protocol. (дата обращения 04.05.2022).

[15] *aSoft-Ltd/files*. URL: https://github.com/aSoft-Ltd/files. (дата обращения 04.05.2022).

[16] *kotlinx-fs*. URL: https://github.com/qwwdfsad/kotlinx-fs. (дата обращения 04.05.2022).

[17] *supernatural-fs*. URL: https://github.com/suparngp/kotlin-multiplatform-projects. (дата обращения 04.05.2022).

[18] *korio*. URL: https://github.com/korlibs/korio. (дата обращения 04.05.2022).

[19] *Korlibs*. URL: https://docs.korge.org/. (дата обращения 04.05.2022).

[20] *okio*. URL: https://github.com/square/okio. (дата обращения 04.05.2022).

[21] *multiplatform-settings*. URL: https://github.com/russhwolf/multiplatform-settings. (дата обращения 04.05.2022).

[22] *KVault*. URL: https://github.com/Liftric/KVault. (дата обращения 04.05.2022).

[23] *Kissme*. URL: https://github.com/netguru/Kissme. (дата обращения 04.05.2022).

[24] *Multiplatform-Preferences*. URL: https://github.com/florent37/Multiplatform-Preferences. (дата обращения 04.05.2022).

[25] *Kotlin Data Storage*. URL: https://github.com/kotlingang/kds. (дата обращения 04.05.2022).

[26] *Delegated properties | Kotlin*. URL: https://kotlinlang.org/docs/delegated-properties.html. (дата обращения 04.05.2022).

[27] *asoft-storage*. URL: https://github.com/andylamax/asoft-storage. (дата обращения 04.05.2022).

[28] *kached*. URL: https://github.com/faogustavo/kached. (дата обращения 04.05.2022).

[29] *Amazon S3. Cloud Object Storage*. URL: https://aws.amazon.com/s3/. (дата обращения 04.05.2022).

[30] *kotlinx-io*. URL: https://github.com/Kotlin/kotlinx-io. (дата обращения 04.05.2022).

[31] *kotlinx-io. Is this library active/supported?* URL: https://github.com/Kotlin/kotlinx-io/issues/54. (дата обращения 04.05.2022).

[32] *Ktor Framework*. URL: https://ktor.io/. (дата обращения 04.05.2022).

[33] *Tinlok*. URL: https://github.com/Fuyukai/Tinlok. (дата обращения 04.05.2022).

[34] *mutifs*. URL: https://github.com/vsalavatov/multifs. (дата обращения 04.05.2022).

[35] *Window.localStorage - Интерфейсы Web API | MDN*. URL: https://developer.mozilla.org/ru/docs/Web/API/Window/localStorage. (дата обращения 10.05.2022).

[36] *Representational state transfer - Wikipedia*. URL: https://en.wikipedia.org/wiki/Representational_state_transfer. (дата обращения 11.05.2022).

[37] *Files and folders overview | Drive API | Google Developers*. URL: https://developers.google.com/drive/api/guides/about-files. (дата обращения 11.05.2022).

[38] *API Яндекс.Диска*. URL: https://yandex.ru/dev/disk/api/concepts/about.html. (дата обращения 11.05.2022).

[39] *Dropbox API v2*. URL: https://www.dropbox.com/developers/documentation/http/documentation. (дата обращения 11.05.2022).

[40] *Amazon Simple Storage Service Documentation*. URL: https://docs.aws.amazon.com/s3/index.html. (дата обращения 11.05.2022).

[41] *Case Sensitivity | Microsoft Docs*. URL: https://docs.microsoft.com/en-us/windows/wsl/case-sensitivity. (дата обращения 11.05.2022).

[42] *Kotlin. Properties. Getters and setters*. URL: https://kotlinlang.org/docs/properties.html#getters-and-setters. (дата обращения 13.05.2022).

[43] *Kotlin. Сопрограммы*. URL: https://kotlinlang.ru/docs/reference/coroutines.html. (дата обращения 13.05.2022).

[44] *Kotlin. Generics*. URL: https://kotlinlang.org/docs/generics.html. (дата обращения 14.05.2022).

[45] *KT-13108. Denotable union and intersection types*. URL: https://youtrack.jetbrains.com/issue/KT-13108/Denotable-union-and-intersection-types#focus=Comments-27-5474923.0-0. (дата обращения 15.05.2022).

[46] *Google Cloud Platform*. URL: https://console.cloud.google.com. (дата обращения 15.05.2022).

[47] *Using OAuth 2.0 to Access Google APIs*. URL: https://developers.google.com/identity/protocols/oauth2. (дата обращения 16.05.2022).

[48] *Android KTX*. URL: https://developer.android.com/kotlin/ktx. (дата обращения 16.05.2022).

[49] *Set up Google Play services*. URL: https://developers.google.com/android/guides/setup. (дата обращения 16.05.2022).

[50] *Promise - JavaScript | MDN*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise. (дата обращения 22.05.2022).

[51] *Inline functions. Reified type parameters*. URL: https://kotlinlang.org/docs/inline-functions.html#reified-type-parameters. (дата обращения 23.05.2022).

[52] *Gradle Build Tool*. URL: https://gradle.org/. (дата обращения 17.05.2022).

[53] *Multieditor*. URL: https://github.com/vsalavatov/multieditor. (дата обращения 22.05.2022).

[54] *Compose Multiplatform, by JetBrains*. URL: https://github.com/JetBrains/compose-jb. (дата обращения 24.05.2022).

[55] *VerticalScrollbar wrong behavior in AlertDialog*. URL: https://github.com/JetBrains/compose-jb/issues/976. (дата обращения 24.05.2022).

[56] *Using LazyColumn in AlertDialog causes IllegalStateException*. URL: https://github.com/JetBrains/compose-jb/issues/1111. (дата обращения 24.05.2022).

[57] *gdrive-cli*. URL: https://github.com/vsalavatov/gdrive-cli. (дата обращения 22.05.2022).

A. Снимки экрана приложения Multieditor

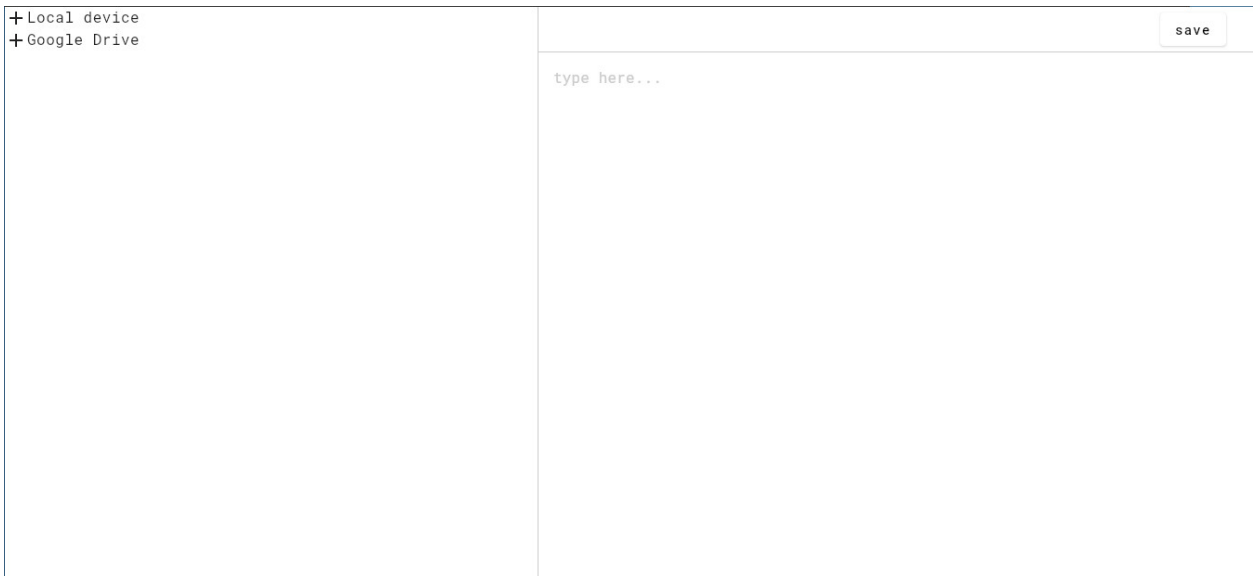

Рис. 7: Multieditor JVM сразу после запуска.

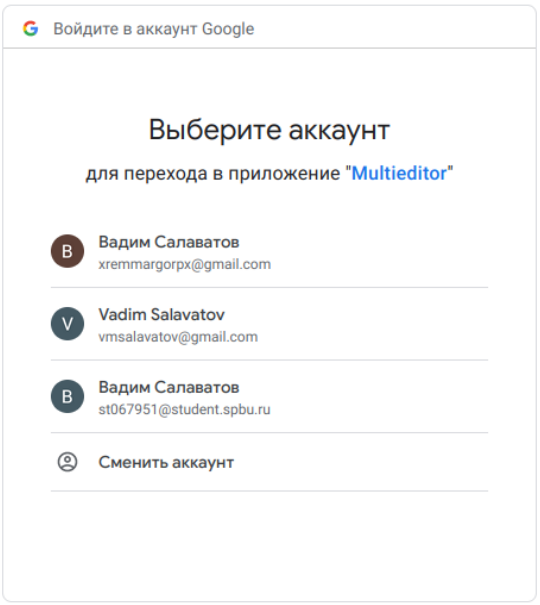

Рис. 8: Процесс подтверждения авторизации на сервисе Google, снимок экрана вкладки в веб-браузере (Multieditor JVM).

Рис. 9: Multieditor JVM в процессе использования с подключенным хранилищем Google Drive.

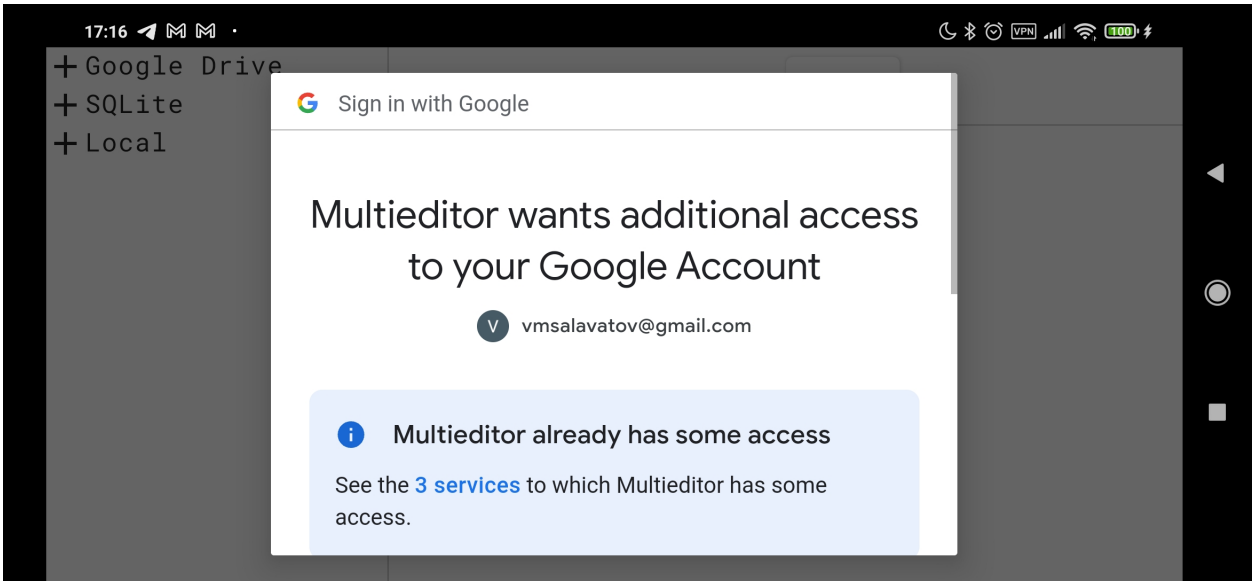

Рис. 10: Процесс подтверждения авторизации на сервисе Google (Multieditor Android).

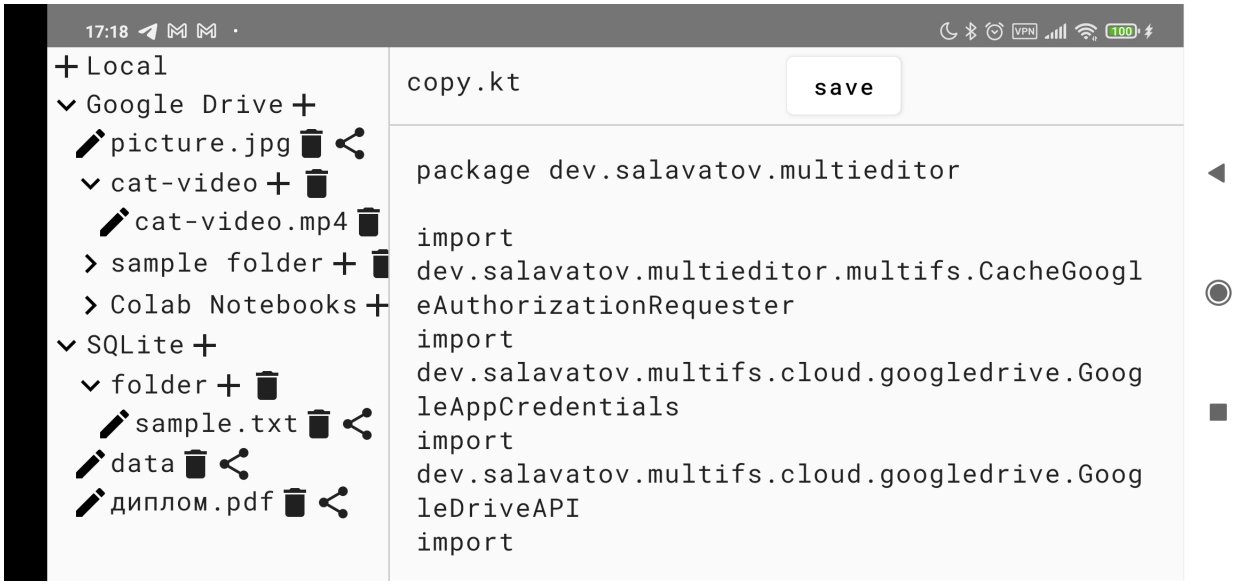

Рис. 11: Снимок экрана Multieditor Android с подключенными хранилищами Google Drive и SQLite.

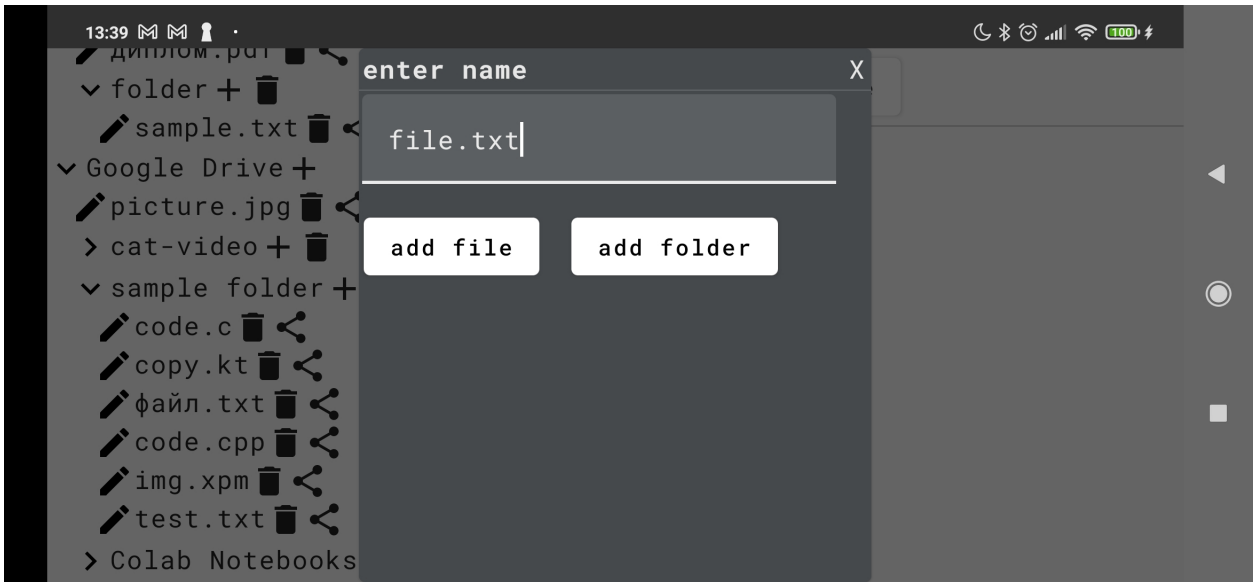

Рис. 12: Всплывающее диаологовое окно Multieditor Android для добавления файла или папки.

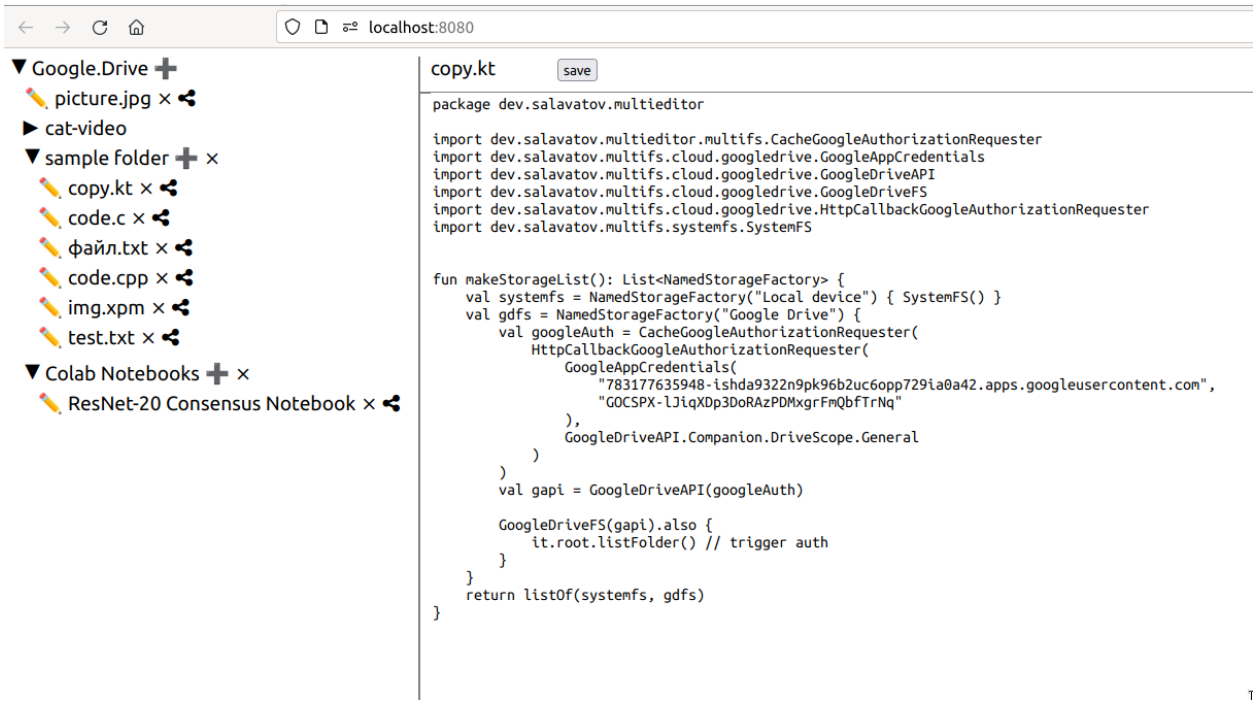

Рис. 13: Снимок экрана Multieditor Web в веб-браузере.