

Geekbrains

**Разработка мобильного приложения "GB Notes" для платформы
Android, представляющее собой редактор заметок.**

Программа: Разработчик - Программист

Специализация: Android-разработка

ФИО: Лысков Никита Алексеевич

Санкт-Петербург

2024 г.

Дипломный проект
" Разработка мобильного приложения "GB Notes" для платформы
Android, представляющее собой редактор заметок".

Содержание

Содержание

| | |
|---------------------------------------------------------------------------------|----|
| Введение..... | 3 |
| Глава 1. Обзор мобильных приложений для создания заметок | 5 |
| 1.1. Что такое мобильное приложение для создания заметок, его особенности | 5 |
| 1.2. Анализ существующих приложений для создания заметок | 6 |
| 1.3. Определение требований к новому приложению | 8 |
| Глава 2. Разработка мобильного приложения “Заметки” | 10 |
| 2.1. Создание нового проекта в Android Studio | 10 |
| 2.2. Разработка основного функционала приложения | 10 |
| 2.3. Тестирование приложения на эмуляторах и реальном устройстве... | 44 |
| Глава 3. Анализ результатов и предложения по улучшению..... | 45 |
| 3.1 Обзор полученных результатов..... | 45 |
| 3.2 Предложения по улучшению приложения | 45 |
| Заключение..... | 46 |
| Список использованной литературы..... | 47 |
| Приложение 1 | 48 |
| Приложение 2..... | 49 |
| Приложение 3..... | 51 |
| Приложение 4..... | 52 |
| Приложение 5..... | 53 |
| Приложение 6..... | 55 |

Введение

В эпоху цифровизации, когда потоки информации непрерывно увеличиваются, способность эффективно управлять этими данными становится критически важной для поддержания продуктивности и организованности. В контексте мобильных технологий, где каждый момент может принести новую идею или задачу, необходимость в надежном и удобном инструменте для записи и управления заметками ощущается особенно остро. Именно поэтому я инициировал разработку мобильного приложения “GB Notes” для платформы Android, целью которого является предоставление пользователям максимально удобного и функционального сервиса для работы с текстовой информацией.

Тема проекта выбрана на основе наблюдения за растущей потребностью в инструментах, которые позволяют пользователям быстро и эффективно записывать и организовывать поступающую информацию, будь то список предстоящих покупок, название интересного фильма, который хотелось бы посмотреть чуть позже или просто краткая и важная информация. Проведя анализ рынка, я обнаружил, что, несмотря на наличие множества приложений для заметок, многие из них не предлагают достаточной гибкости или функциональности, чтобы удовлетворить потребности пользователей или имеют сложный интерфейс.

Цель проекта - разработать интуитивно понятное и функциональное приложение, которое будет решать проблему хранения и поиска заметок. Приложение будет предлагать функции создания, редактирования и удаления заметок, а также создания папок для их организации. Это позволит пользователям легко управлять своими заметками и быстро находить нужную информацию.

Для выполнения текущего проекта планируется использование Kotlin для Android и Android Studio как основной среды разработки.

План работы включает в себя следующие этапы:

1. Анализ требований к функционалу приложения.

2. Проектирование интерфейса пользователя.
3. Реализация функционала приложения.
4. Тестирование приложения на эмуляторах и реальном устройстве.
5. Отладка и оптимизация приложения.

В дальнейшем планируется расширение функционала приложения, включая добавление функций поиска по содержимому заметок, форматирования текста, закрепления заметок в списке, скрытия заметок в раздел "скрытые" и создания задач с функцией установки напоминаний.

В целом, этот проект представляет собой важный шаг в области разработки мобильных приложений, поскольку он направлен на удовлетворение конкретных потребностей пользователей и улучшение их продуктивности и организации информации. Я надеюсь, что моё приложение “GB Notes” будет полезным инструментом для многих пользователей и поможет им в их повседневной жизни и работе. А также позволит мне погрузиться в детали разработки приложения, для последующей реализации в новых проектах.

Глава 1. Обзор мобильных приложений для создания заметок

1.1 Что такое мобильное приложение для создания заметок, его особенности

Мобильные приложения для создания заметок — это категория программного обеспечения, предназначенная для удобства записи и хранения информации в цифровом формате. Они обеспечивают пользователей инструментами для фиксации мыслей, идей, списков дел и других важных данных, которые могут быть легко доступны в любое время и в любом месте. Эти приложения часто используются для управления личными задачами, планирования рабочих проектов, составления списков покупок и сохранения важных заметок, которые могут включать текст, изображения и даже аудиозаписи.

Основные характеристики мобильных приложений для создания заметок включают:

- 1) **Интуитивно понятный интерфейс:** Приложения разработаны таким образом, чтобы минимизировать кривую обучения и позволить пользователям быстро начать работу.
- 2) **Синхронизация данных:** Большинство приложений предлагают облачную синхронизацию, позволяющую пользователям доступ к их заметкам на различных устройствах.
- 3) **Интеграция:** Многие приложения предоставляют возможность интеграции с другими сервисами, такими как календари, электронная почта и социальные сети.
- 4) **Безопасность:** Защита конфиденциальности и безопасность данных являются ключевыми аспектами, с функциями шифрования и парольной защиты.
- 5) **Персонализация:** Пользователи могут настраивать внешний вид и ощущение своих заметок, выбирая темы, шрифты и цветовые схемы.

- 6) **Расширенные функции:** Некоторые приложения предлагают дополнительные инструменты, такие как распознавание рукописного текста, голосовые заметки и интеграцию с внешними устройствами.

1.2 Анализ существующих приложений для создания заметок

Для анализа популярных приложений для создания заметок, мы рассмотрим следующие аспекты:

- 1) **Функциональность:** Какие основные и дополнительные функции предлагает каждое приложение?
- 2) **Пользовательский интерфейс:** насколько удобен и понятен интерфейс приложения?
- 3) **Производительность:** как быстро работает приложение, и насколько оно стабильно?
- 4) **Отзывы пользователей:** Какие плюсы и минусы отмечают пользователи в своих отзывах?
- 5) **Ценовая политика:** Является ли приложение бесплатным, или предлагает ли оно платные подписки с расширенными функциями?

Этот анализ поможет нам лучше понять потребности пользователей и определить, какие функции должны быть включены в наше приложение.

Рассмотрим такие приложения как: "Evernote", "Google Keep" и "OneNote".

Evernote:

1. **Функциональность:** Evernote предлагает широкий спектр функций, включая возможность создания заметок, списков, веб-клиппингов и аудиозаписей. Пользователи могут организовывать заметки в блокноты и тегировать их для удобства поиска. Однако, некоторые пользователи отмечают, что из-за обилия функций приложение может показаться перегруженным.

2. **Пользовательский интерфейс:** Интерфейс Evernote хорошо структурирован, но из-за большого количества функций новым пользователям может потребоваться время, чтобы привыкнуть к нему. Некоторые пользователи считают интерфейс несколько устаревшим.

3. **Производительность:** Приложение работает стабильно, но иногда пользователи сталкиваются с задержками при синхронизации больших объемов данных.

4. **Отзывы пользователей:** Большинство отзывов положительные, особенно в части функциональности и надежности. Однако, некоторые пользователи жалуются на высокую стоимость подписки и сложность в освоении всех функций.

5. **Ценовая политика:** Evernote предлагает несколько уровней подписки, включая бесплатный план. Платные планы предоставляют дополнительные функции, такие как больше места для хранения и более продвинутые инструменты организации.

Google Keep:

1. **Функциональность:** Google Keep ориентирован на простоту и скорость, позволяя пользователям быстро создавать заметки и списки. Однако, в отличие от Evernote, Keep не предлагает расширенные инструменты организации, такие как блокноты или теги.

2. **Пользовательский интерфейс:** Интерфейс Google Keep очень простой и цветной, что делает приложение легким для начала работы. Тем не менее, некоторые пользователи находят его слишком ограниченным для сложной организации заметок.

3. **Производительность:** Keep работает очень быстро и редко вызывает технические проблемы. Однако, приложение иногда может быть слишком базовым для пользователей, нуждающихся в более мощных функциях.

4. **Отзывы пользователей:** Пользователи ценят Google Keep за его простоту и интеграцию с другими продуктами Google. В то же время, отсутствие

некоторых функций, таких как форматирование текста, может быть недостатком для некоторых пользователей.

5. Ценовая политика: Google Keep является полностью бесплатным, что делает его доступным для широкой аудитории..

OneNote:

1. Функциональность: OneNote предлагает гибкую структуру для создания заметок, которые можно размещать в любом месте на странице. Приложение поддерживает мультимедийный контент и рукописный ввод. Однако, некоторые пользователи считают, что приложение может быть сложным в освоении из-за своей нелинейной структуры.

2. Пользовательский интерфейс: Интерфейс OneNote хорошо организован и предлагает много функций, но может показаться перегруженным для новых пользователей. Также, некоторые пользователи отмечают несоответствие в дизайне между различными платформами.

3. Производительность: OneNote обычно работает хорошо, но может страдать от проблем с синхронизацией, особенно при использовании на нескольких устройствах.

4. Отзывы пользователей: Пользователи высоко оценивают мощные функции OneNote, такие как возможность делиться заметками и интеграция с Office 365. Однако, некоторые пользователи жалуются на проблемы с синхронизацией и сложность интерфейса.

5. Ценовая политика: OneNote доступен бесплатно, но для полного доступа к функциям Office 365 требуется подписка.

1.3 Определение требований к новому приложению

На основе детального анализа существующих приложений для создания заметок и отзывов пользователей, мы можем сделать вывод, что идеальное приложение для заметок должно сочетать в себе простоту использования с

глубокими функциональными возможностями. Пользователи ценят интуитивно понятный интерфейс, но также ожидают от приложения гибкости и мощи для управления большим объемом информации. Следовательно, новое приложение должно учитывать как положительные стороны существующих решений, так и их недостатки, предлагая улучшенный пользовательский опыт.

Требования к новому приложению:

1. Удобство управления заметками: Приложение должно предоставлять легкие в использовании инструменты для создания, редактирования и удаления заметок, обеспечивая при этом гибкость организации через сортировку по разделам и папкам.

2. Мультимедийная поддержка: Возможность прикрепления медиафайлов и документов к заметкам расширит способы использования приложения, позволяя сохранять не только текст, но и визуальные и аудиальные данные.

3. Эффективный поиск: Разработанная система поиска по заметкам и фильтрация, по ключевым словам, должна обеспечивать быстрый доступ к нужной информации без необходимости просматривать все заметки.

4. Напоминания и уведомления: Система уведомлений для напоминаний и сроков выполнения задач поможет пользователям оставаться в курсе важных событий и сроков.

5. Безопасность и доступность данных: Функции резервного копирования и восстановления данных обеспечат безопасность информации и возможность доступа к ней с любого устройства.

Эти требования отражают желание пользователей иметь универсальное приложение, которое будет одинаково удобно как для быстрых заметок, так и для сложной организации информации. Новое приложение должно предложить баланс между простотой и функциональностью, учитывая современные тенденции и ожидания пользователей. Основываясь на этих требованиях, мы можем приступить к следующему этапу проекта — проектированию и разработке приложения.

Глава 2. Разработка мобильного приложения “Заметки”

2.1 Создание нового проекта в Android Studio

Создание нового проекта в Android Studio является фундаментальным этапом, который задает тон всему процессу разработки. Этот этап начинается с запуска Android Studio и выбора опции “Start a new Android Studio project”. Здесь важно выбрать имя проекта, которое будет отражать его функциональность и удобно восприниматься пользователями. Например, для приложения “Заметки” подходящим именем может быть “QuickNotes” или “MemoPad”.

Далее, необходимо определить минимальную версию Android (API level), которую будет поддерживать приложение. Это важно для обеспечения совместимости приложения с большинством устройств пользователей. Рекомендуется выбирать версию, которая охватывает как минимум 90% устройств на рынке.

После этого следует выбор шаблона активности. Для приложения заметок подойдет шаблон “Empty Activity”, который предоставляет чистый холст для дальнейшей разработки. Это позволит начать с основ и постепенно добавлять необходимые элементы интерфейса и функциональности.

2.2 Разработка основного функционала приложения

Разработка основного функционала приложения “Заметки” включает в себя несколько ключевых шагов:

Шаг 1: Разработка пользовательского интерфейса

В начале моего пути к созданию приложения “Заметки”, я сосредоточился на разработке пользовательского интерфейса, который был бы не только функциональным, но и приятным в использовании. Следуя принципам Material

Design, я стремился создать чистый и современный вид, который облегчал бы пользователю взаимодействие с приложением.

1. Главный экран приложения.

На главном экране моего приложения я решил разместить такие элементы, как: Список заметок, список папок, кнопку для перехода на страницу создания заметки и кнопку для перехода к списку папок заметок.

FloatingActionButton: Я добавил FloatingActionButton в основной экран, чтобы пользователи могли легко добавлять новые заметки. Эта кнопка была размещена в нижнем правом углу экрана, делая её доступной, но не мешающей просмотру списка заметок (см. Рисунок 2.1).

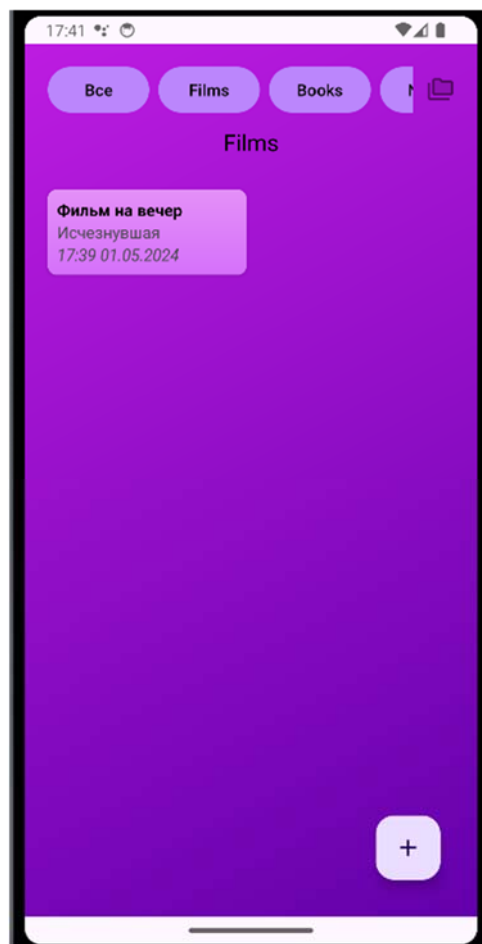


Рисунок 2.1 Размещение элементов пользовательского интерфейса на главном экране.

RecyclerView: Для отображения списка заметок я использовал RecyclerView, что позволило мне создать динамичный и гибкий список, который

мог бы эффективно работать с большим количеством элементов (см. Рисунок 2.2).

RecyclerView: Для отображения списка папок на главном экране, я так же использовал RecyclerView, для удобства автоматического обновления списка. (см. Рисунок 2.2).

ImageButton: Для возможности перехода на экран создания и удаления папок с заметками, достаточно простой кнопки. Её я разместил в правом верхнем углу в конце RecyclerView с папками, для интуитивно-понятного взаимодействия (см. Рисунок 2.2).

TextView: Для отображения текущей выбранной папки я использовал обычный TextView. Расположен он по середине прямо под RecyclerView папок (см. Рисунок 2.2).

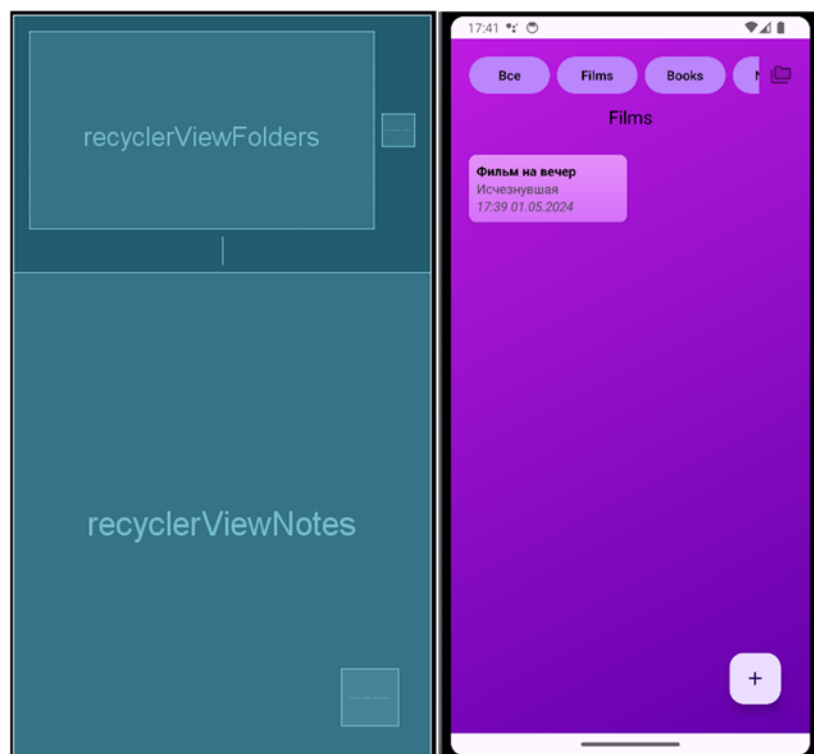


Рисунок 2.2 RecyclerView заметок и папок на главном экране.

Для отображения самой заметки я использовал отдельный layout.

Карточки заметок: Каждая заметка в списке была представлена в виде карточки, содержащей краткий обзор и дату последнего изменения, что позволяет пользователю быстро находить нужную информацию. Число

выводимых строк содержимого заметки было ограничено пятью с целью более аккуратного представления их в списке, а так же, что бы не было ситуации, когда одна заметка занимала бы очень большую часть пространства списка (см. Рисунок 2.3).

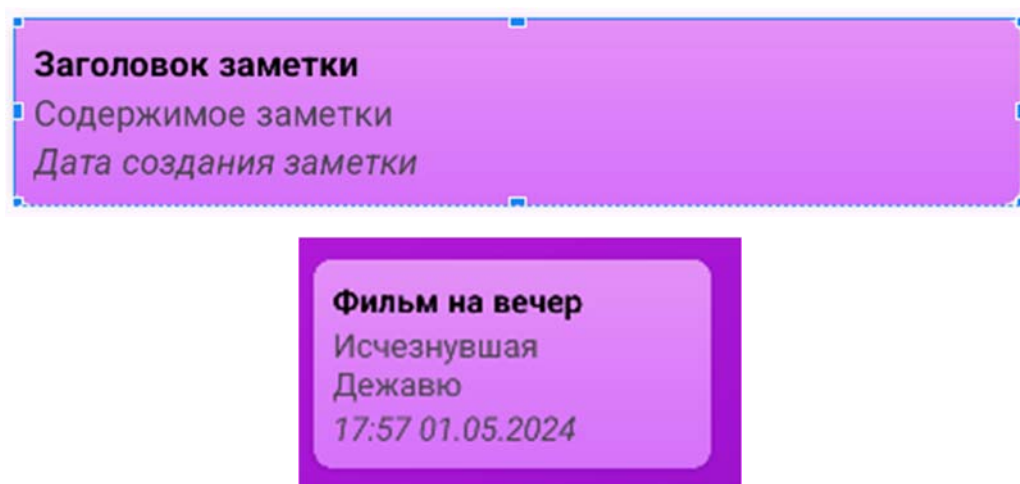


Рисунок 2.3 Структура карточки заметок для главного экрана

Элементы папок для RecyclerView были сделаны в виде простых **Button**, содержащих в себе название папки (см. Рисунок 2.4).

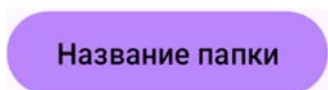
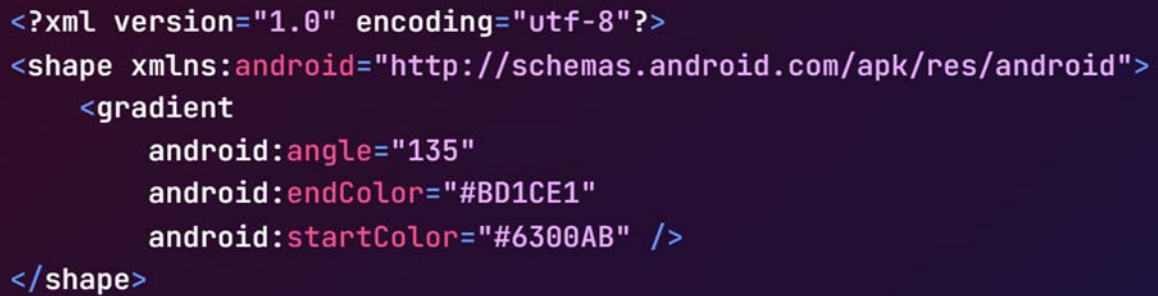


Рисунок 2.4 Элемент кнопки с названием папки

Как можно было заметить, для заднего фона главного экрана был выбран градиент, это отдельный кастомный ресурс, представляющий собой код .xml (см. Рисунок 2.5).



```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android">
  <gradient
    android:angle="135"
    android:endColor="#BD1CE1"
    android:startColor="#6300AB" />
</shape>
```

Рисунок 2.5 Код слоя для заднего фона MainFragment

2. Экран создания/редактирования заметок.

При разработке пользовательского интерфейса для экрана создания заметки, я сосредоточился на обеспечении плавного и интуитивно понятного процесса ввода данных. Экран должен был быть не только функциональным, но и эстетически приятным, чтобы пользователи чувствовали удовольствие от процесса создания своих заметок.

Toolbar: Я начал с добавления Toolbar, который действовал как верхняя панель приложения. Это было важно для обеспечения консистентности интерфейса по всему приложению. В Toolbar я разместил кнопки для навигации: кнопку “Назад” для возврата к предыдущему экрану и кнопку “Готово” для сохранения заметки (см. Рисунок 2.6).

Выбор папки: Для удобства организации заметок я добавил выпадающий список `AutoCompleteTextView` внутри `TextInputLayout`, который позволял пользователям выбирать папку для новой заметки из предварительно определенного списка (см. Рисунок 2.6).

Поля ввода: Я создал поля ввода для заголовка и содержимого заметки. Эти поля были реализованы с помощью `EditText`, что обеспечивало простой и чистый интерфейс для ввода текста (см. Рисунок 2.6).

Дата и время: Я также включил элемент TextView для отображения даты и времени создания или последнего изменения заметки, что добавляло контекст и помогало в организации (см. Рисунок 2.6).

Текущая папка для сохранения заметки: Я добавил TextView для отображения текущей выбранной папки, чтобы пользователю было удобно определить, в какую папку будет сохранена его заметка после нажатия на кнопку в виде галочки. Расположение этого TextView было выбрано по центру Toolbar (см. Рисунок 2.7).

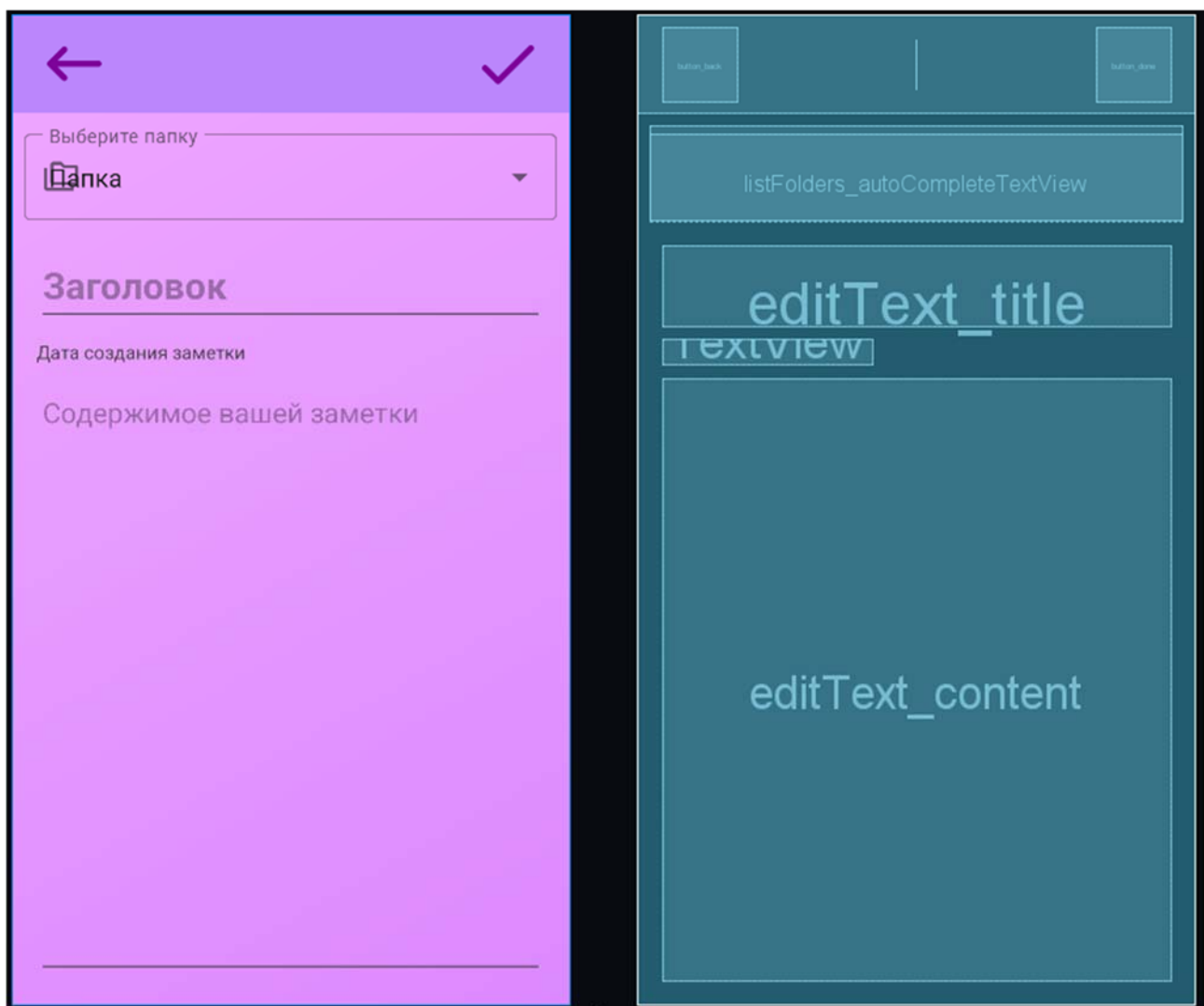


Рисунок 2.6 Разметка фрагмента создания заметки (CreateNoteFragment)

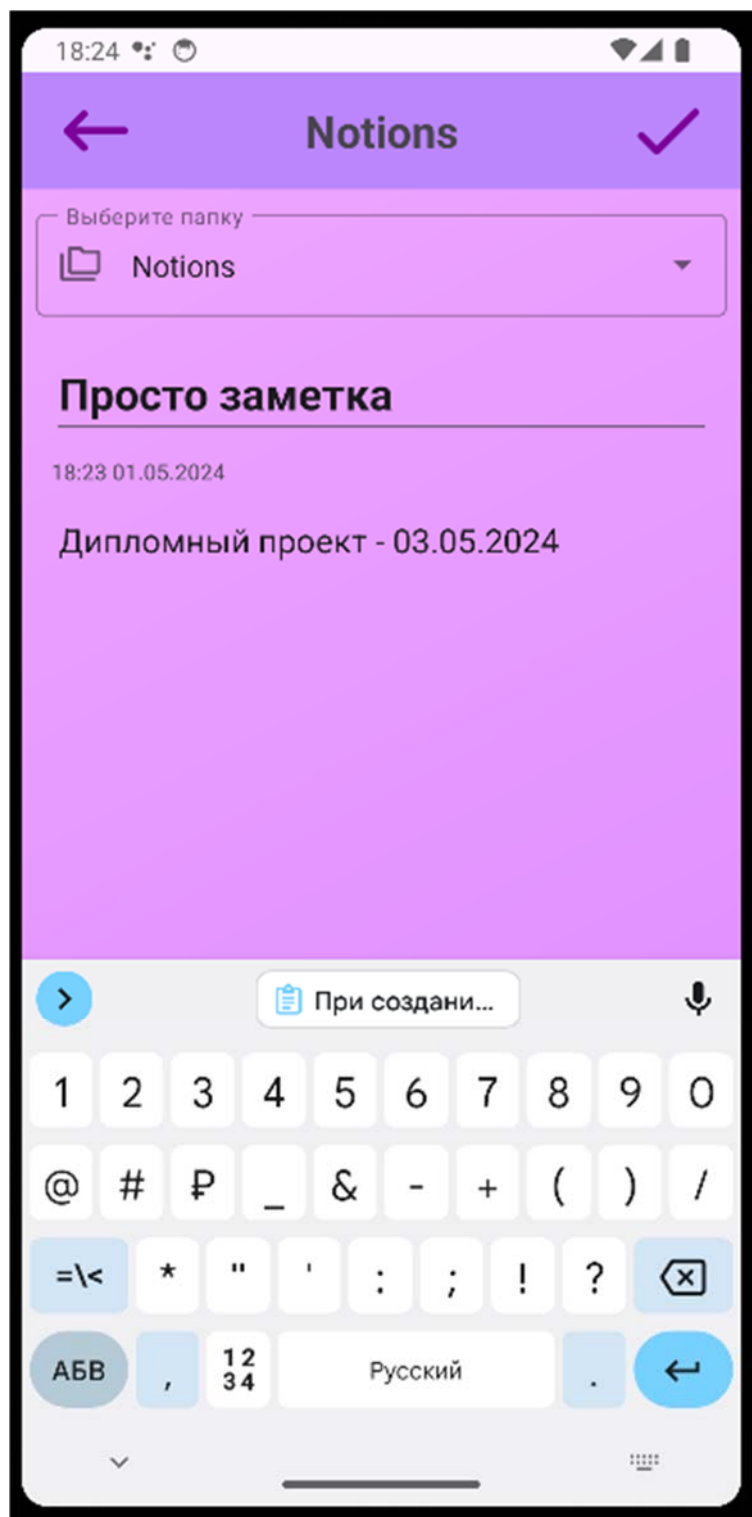


Рисунок 2.7 Экран создания заметок в эмуляторе

3. Экран списка папок.

При создании экрана списка папок для моего приложения “Заметки”, я стремился к тому, чтобы пользователи могли легко и удобно управлять своими

категориями заметок. Экран должен был быть интуитивно понятным и функциональным, обеспечивая эффективное взаимодействие с приложением.

AppBarLayout и Toolbar: Я начал с размещения AppBarLayout и Toolbar, чтобы создать унифицированный верхний колонтитул по всему приложению. В Toolbar были добавлены элементы управления, такие как кнопка “Назад” для возвращения к предыдущему экрану и заголовок экрана, который явно указывал на текущую функцию — управление папками (см. Рисунок 2.8).

RecyclerView: Для отображения списка папок я использовал RecyclerView, настроенный с LinearLayoutManager, что обеспечивало простой и понятный вертикальный список папок. Это позволяло пользователям легко просматривать и выбирать нужные категории (см. Рисунок 2.8).

FloatingActionButton: Я добавил FloatingActionButton в нижнюю правую часть экрана, предоставляя пользователям быстрый доступ к функции создания новой папки. Эта кнопка была важна для обеспечения возможности быстрого добавления новых категорий без необходимости переходить в другие меню или экраны (см. Рисунок 2.8).

Дизайн и стили: Я тщательно подобрал цвета и стили для элементов интерфейса, чтобы они соответствовали общей цветовой схеме приложения и были приятны для глаз. Фон экрана и элементы управления были оформлены таким образом, чтобы создать гармоничный и современный вид.

Этот экран был разработан с учетом удобства и эффективности, чтобы пользователи могли без труда организовывать свои заметки и повышать свою продуктивность.

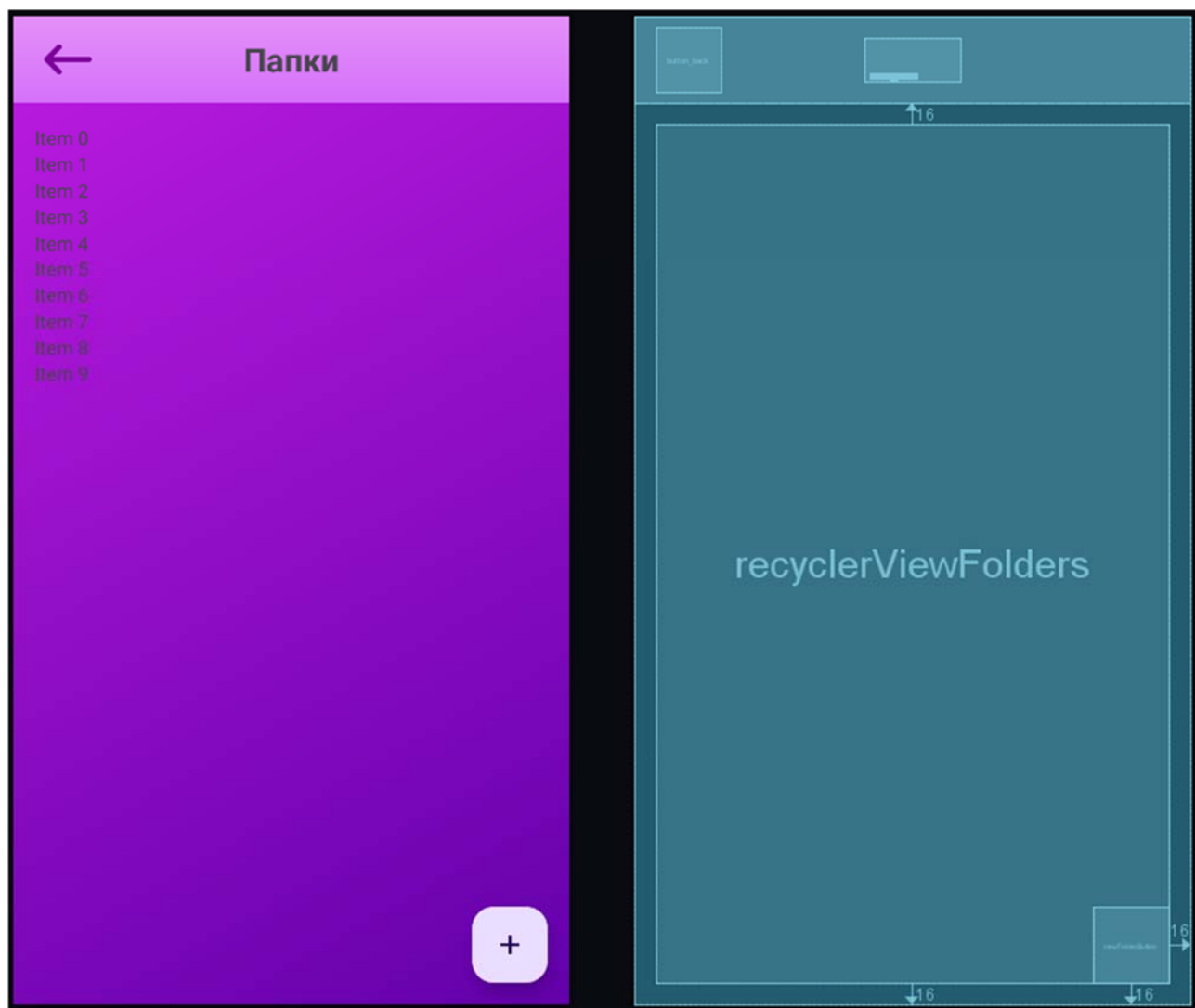


Рисунок 2.8 Экран создания папок для заметок

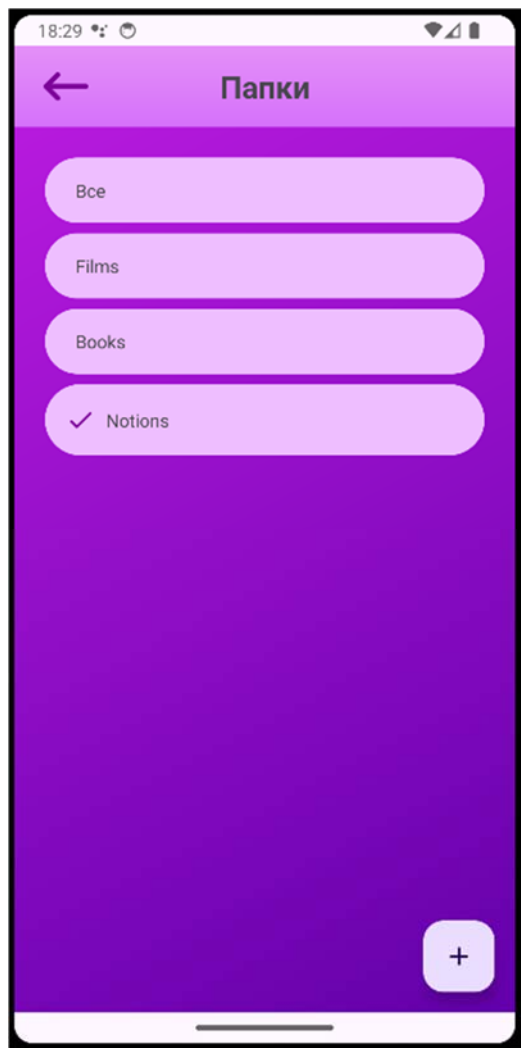


Рисунок 2.9 Экран создания папок в эмуляторе

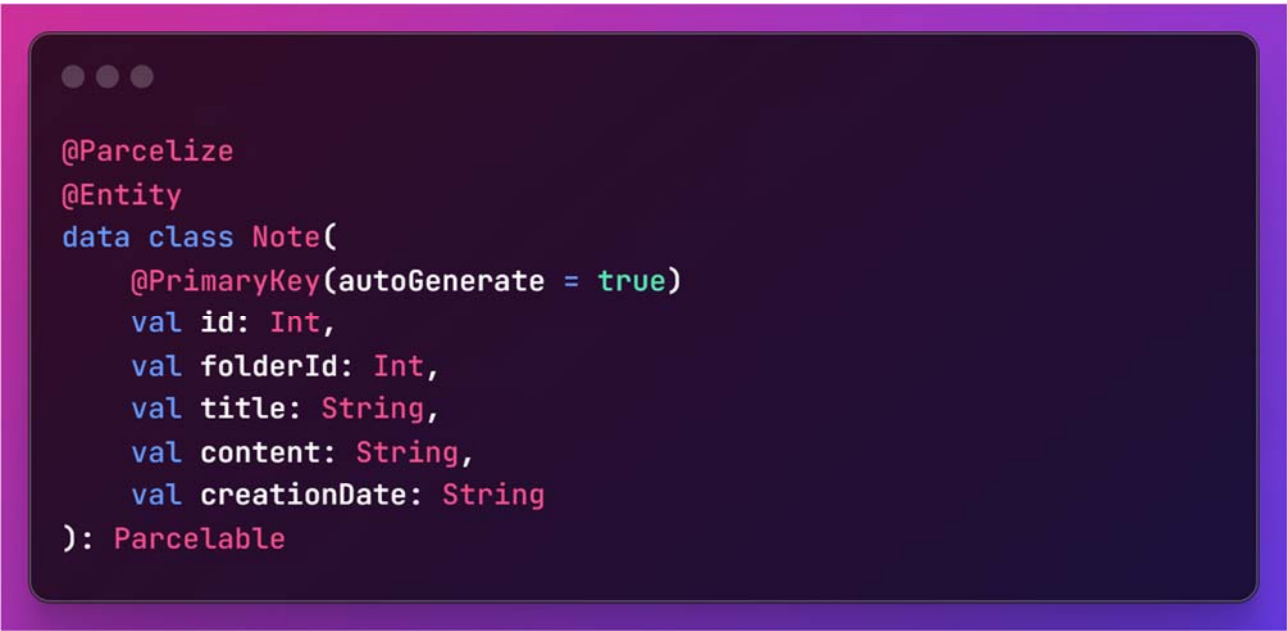
Шаг 2: Программирование основных операций с заметками

Этот шаг является сердцем функциональности приложения.

В самом начале, была разработана будущая структура самого приложения, его архитектура и определены необходимые классы для работы данной системы.

В первую очередь были разработаны класс самой сущности заметки, а также классы взаимодействия с базой данных. Для хранения заметок я выбрал использование Room Database, которая предоставляла мощный и гибкий способ работы с локальной базой данных.

Класс Note в пакете model представляет собой Data class (см. Рисунок 2.10)

A screenshot of a code editor with a dark background and purple/pink accents. The code is written in Kotlin and defines a data class named 'Note'. The class is annotated with '@Parcelize' and '@Entity'. It has a primary key 'id' and five other properties: 'folderId', 'title', 'content', and 'creationDate'. The class implements the 'Parcelable' interface.

```
@Parcelize
@Entity
data class Note(
    @PrimaryKey(autoGenerate = true)
    val id: Int,
    val folderId: Int,
    val title: String,
    val content: String,
    val creationDate: String
): Parcelable
```

Рисунок 2.10 Класс Note

В этом классе определена сущность заметки с такими полями как:

- 1) id – идентификатор заметки,
- 2) folderId – идентификатор папки, в которую помещается заметка,
- 3) title – заголовок заметки,
- 4) content – содержимое заметки,
- 5) creationDate – время и дата создания заметки.

Далее был создан интерфейс NoteDao.

Я реализовал NoteDao со следующими функциями (см. Рисунок 2.11).

```

@Dao
interface NoteDao {

    @Query("SELECT * FROM note ORDER BY creationDate DESC")
    suspend fun getAllNotes(): List<Note>

    @Query("SELECT * FROM note WHERE folderId = :folderId ORDER BY creationDate DESC")
    suspend fun getNotesByFolder(folderId: Int): List<Note>

    @Query("SELECT * FROM note WHERE id = :noteId")
    suspend fun getNoteById(noteId: Int): Note

    @Insert
    suspend fun insert(note: Note)

    @Update
    suspend fun update(note: Note)

    @Delete
    suspend fun delete(note: Note)

    @Query("DELETE FROM note WHERE folderId = :folderId")
    suspend fun deleteNotesInFolder(folderId: Int)
}

```

Рисунок 2.11 Интерфейс NoteDao

1) *getAllNotes()* – функция позволяет получить список заметок из таблицы *note*. Дополнительно указана сортировка по дате создания.

2) *getNotesByFolder(folderId: Int)* – функция позволяет получить список заметок, вложенных в определенную папку по её идентификатору.

3) *getNoteById(noteId: Int)* – функция позволяет получить из репозитория заметку по её идентификатору.

4) функции *insert*, *update*, *delete* -стандартные функции по вставке, обновлению и удалению элементов, в нашем случае – заметок в репозитории.

5) *deleteNotesInFolder(folderId: Int)* – функция для удаления всех заметок в определенной папке.

Затем был создан репозиторий.

Я определил *NoteRepository*, который служил связующим звеном между *NoteDao* и пользовательским интерфейсом, обеспечивая разделение логики и представления (см. Рисунок 2.12).

```

class NoteRepository @Inject constructor(
    private val noteDao: NoteDao,
) {

    // Получить все заметки
    suspend fun getAllNotes() = noteDao.getAllNotes()

    // Получить заметку по id
    suspend fun getNoteById(noteId: Int) =
noteDao.getNoteById(noteId)

    // Вставить новую заметку
    suspend fun insert(note: Note) {
        noteDao.insert(note)
    }

    // Обновить существующую заметку
    suspend fun update(note: Note) = noteDao.update(note)

    // Удалить заметку
    suspend fun delete(note: Note) {
        noteDao.delete(note)
    }

    // Получить заметки из выбранной папки
    suspend fun getNotesBySelectedFolder(folderId: Int): List<Note>
{
    return noteDao.getNotesByFolder(folderId)
}
}

```

Рисунок 2.12 Класс NoteRepository

Оставленные в коде комментарии подскажут, для чего используются каждая из функций в этом классе.

Сам класс базы данных был реализован в виде наследника RoomDatabase (см. Рисунок 2.13).

```

@Database(entities = [Note::class, Folder::class], version = 2)
abstract class AppDatabase : RoomDatabase() {
    abstract fun noteDao(): NoteDao
    abstract fun folderDao(): FolderDao
}

```

Рисунок 2.13 Абстрактный класс базы данных AppDatabase

Версия на данный момент =2, так как было обновление компонентов в базе данных.

Теперь, для возможности реализации такого функционала, как распределение заметок по папкам, необходимо аналогичным образом реализовать другие классы нашего приложения.

Реализация возможности создания папок для заметок — это важная часть разработки приложения для заметок. Это позволяет пользователям организовывать свои заметки по категориям или темам, что делает поиск и доступ к заметкам более удобными.

Для этого я создал класс Folder в пакете model моего проекта (см. Рисунок 2.14)

```

@Parcelize
@Entity
data class Folder(
    @PrimaryKey(autoGenerate = true)
    val id: Int?,
    val name: String,
    var isSelected: Boolean = false
): Parcelable

```

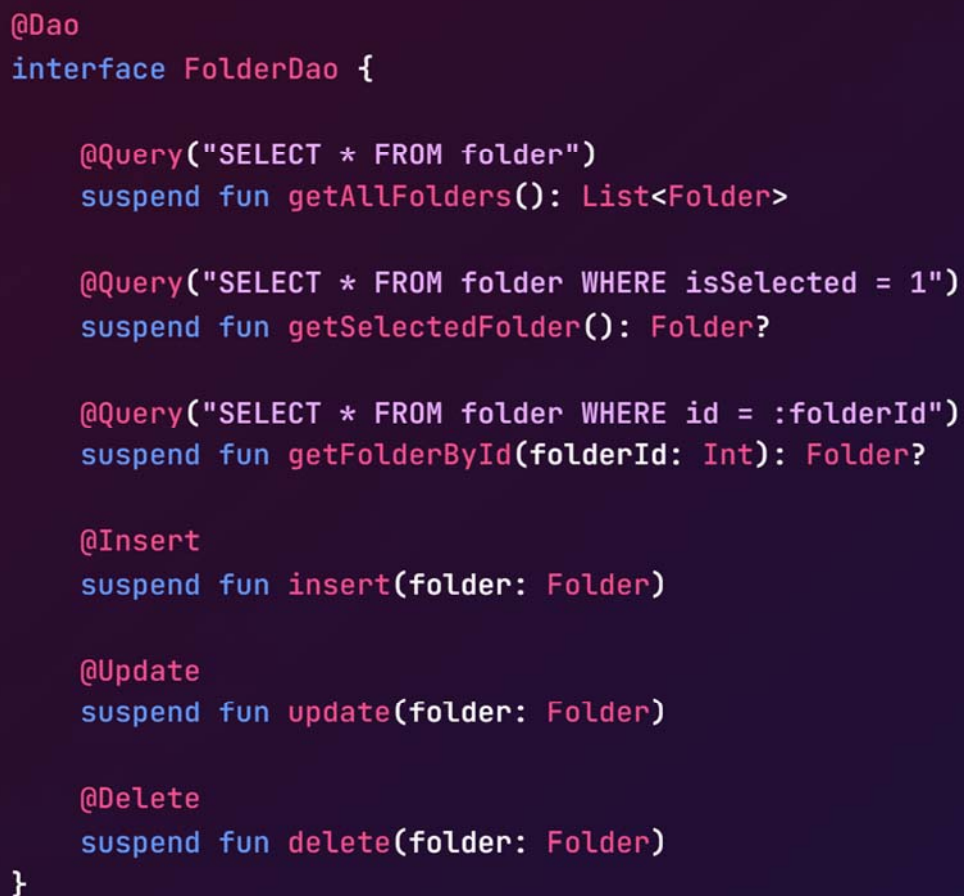
Рисунок 2.14 Data-класс Folder

В этом классе я определил следующие поля:

- 1) id – уникальный идентификатор папки,
- 2) name – имя папки,
- 3) isSelected – "флаг", показывающий, выбрана ли эта папка как текущая.

Далее был создан интерфейс FolderDao.

Я реализовал FolderDao со следующими функциями (см. Рисунок 2.15).

A screenshot of a code editor with a dark background and a purple border. The code defines the FolderDao interface with several methods annotated with @Query, @Insert, @Update, and @Delete. The code is as follows:

```
@Dao
interface FolderDao {

    @Query("SELECT * FROM folder")
    suspend fun getAllFolders(): List<Folder>

    @Query("SELECT * FROM folder WHERE isSelected = 1")
    suspend fun getSelectedFolder(): Folder?

    @Query("SELECT * FROM folder WHERE id = :folderId")
    suspend fun getFolderById(folderId: Int): Folder?

    @Insert
    suspend fun insert(folder: Folder)

    @Update
    suspend fun update(folder: Folder)

    @Delete
    suspend fun delete(folder: Folder)
}
```

Рисунок 2.15 Интерфейс FolderDao

1) *getAllFolders()* – функция позволяет получить список всех папок из таблицы folder.

2) *getSelectedFolder()* – функция позволяет получить выбранную текущую папку.

3) *getFolderById(folderId: Int)* – функция позволяет получить из репозитория папку по её идентификатору.

4) функции *insert*, *update*, *delete* -стандартные функции по вставке, обновлению и удалению элементов, в нашем случае – папок в репозитории.

Затем был создан репозиторий папок.

Я определил *FolderRepository*, который служил связующим звеном между *FolderDao* и пользовательским интерфейсом, обеспечивая разделение логики и представления (см. Рисунок 2.16).

```

class FolderRepository @Inject constructor(
    private val folderDao: FolderDao,
    private val noteDao: NoteDao
) {
    // Получить все папки
    suspend fun getAllFolders() = folderDao.getAllFolders()

    // Вставить новую папку
    suspend fun insert(folder: Folder) = folderDao.insert(folder)

    // Обновить существующую папку
    suspend fun update(folder: Folder) = folderDao.update(folder)

    // Получить папку по id
    suspend fun getFolderById(folderId: Int) =
        folderDao.getFolderById(folderId)

    // Удалить папку
    suspend fun deleteFolder(folder: Folder) {
        if (folder.name != "Bce") {
            folderDao.delete(folder)
            noteDao.deleteNotesInFolder(folder.id!!)
        }
    }

    suspend fun setSelectedFolder(folder: Folder) {
        val selectedFolder = getSelectedFolder()
        if (selectedFolder != null) {
            // Сбросить флаг выбранной папки для текущей выбранной папки
            update(selectedFolder.copy(isSelected = false))
        }
        // Установить флаг выбранной папки для новой выбранной папки
        update(folder.copy(isSelected = true))
    }

    // Получить выбранную папку
    private suspend fun getSelectedFolder() = folderDao.getSelectedFolder()
    suspend fun getMainFolder(): Folder? {
        return folderDao.getFolderById(1)
    }
}

```

Рисунок 2.16 Интерфейс FolderDao

Оставленные в коде комментарии подскажут, для чего используются каждая из функций в этом классе.

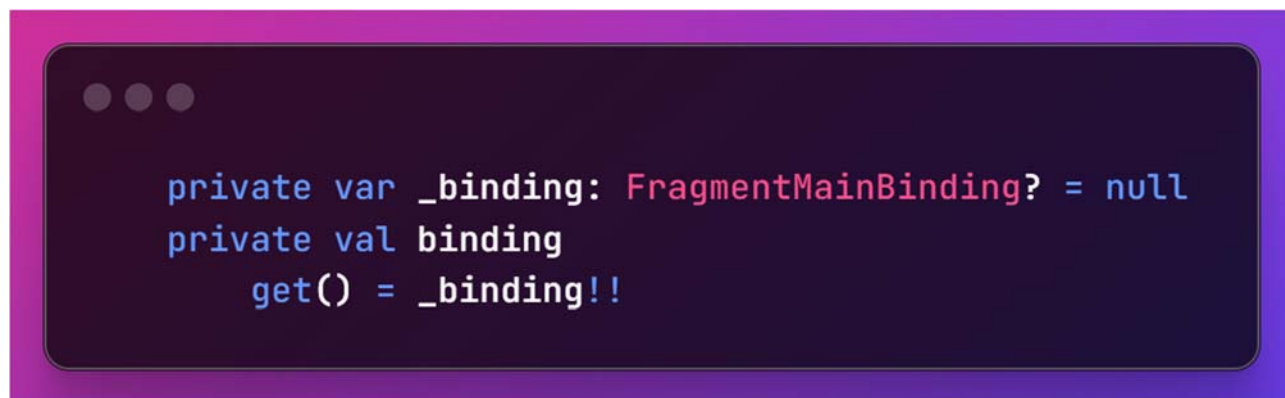
После того, как были созданы базовые сущности и классы, необходимо перейти к созданию основных классов, а именно классу фрагментов и `viewModel`, а также адаптеров для взаимодействия с пользователем.

MainFragment.

Это главный фрагмент. Он же основной экран нашего приложения. Его пользовательский интерфейс был описан в предыдущем разделе, а сейчас я опишу его функциональную часть.

Основной экран служит центральным узлом для доступа к функциям создания, просмотра и управления заметками и папками.

Инициализация и привязка: Я начал с инициализации переменных для привязки данных и управления состоянием фрагмента, используя `FragmentMainBinding` для связывания компонентов пользовательского интерфейса с логикой приложения (см. Рисунок 2.17).



```
private var _binding: FragmentMainBinding? = null
private val binding
    get() = _binding!!
```

Рисунок 2.17 Объявление переменной "FragmentMainBinding"

ViewModel и адаптеры: Для управления данными и бизнес-логикой я использовал `MainViewModel`, который был связан с фрагментом через `ViewModelFactory`. Адаптеры `NoteAdapter` и `FolderAdapter` были настроены для отображения списков заметок и папок соответственно (см. Рисунки 2.18, 2.19 и 2.20).



```
@Inject
lateinit var mainViewModelFactory: ViewModelFactory
private val mainViewModel: MainViewModel by viewModels
{ mainViewModelFactory }
```

Рисунок 2.18 Объявление переменной MainViewModel

```
private lateinit var noteAdapter: NoteAdapter
private lateinit var folderAdapter: FolderAdapter
```

Рисунок 2.19 Объявление NoteAdapter и FolderAdapter

```
noteAdapter = NoteAdapter(
    clickNote = { note -> onNoteClick(note) },
    longClickNote = { note -> onNoteLongClick(note) }
)

folderAdapter = FolderAdapter(onChangeShowNote = { idFolder ->
    viewLifecycleOwner.lifecycleScope.launch {
        mainViewModel.changeListNote(idFolder)
    }
})
```

Рисунок 2.20 Инициализация NoteAdapter и FolderAdapter

RecyclerView: Для отображения заметок и папок я использовал два RecyclerView, каждый из которых был настроен с соответствующими менеджерами компоновки для оптимального отображения элементов (см. Рисунки 2.21 и 2.22).

```
binding.recyclerViewNotes.adapter = noteAdapter

val layoutManagerNotes = StaggeredGridLayoutManager(2, StaggeredGridLayoutManager.VERTICAL)

binding.recyclerViewNotes.layoutManager = layoutManagerNotes
```

Рисунок 2.21 Настройка RecyclerView для заметок

```
binding.recyclerViewFolders.adapter = folderAdapter

val layoutManagerFolders =
    LinearLayoutManager(context, LinearLayoutManager.HORIZONTAL, false)

binding.recyclerViewFolders.layoutManager = layoutManagerFolders
```

Рисунок 2.22 Настройка RecyclerView для папок

Обновление данных: Я реализовал механизмы для обновления списка заметок и папок в реальном времени, используя Flow и LiveData, что позволяло отслеживать изменения в базе данных и немедленно отображать их на экране (см. Рисунки 2.23 и 2.24).

```
// Обновляем список заметок при изменении данных
viewLifecycleOwner.lifecycleScope.launch {
    mainViewModel.allNotesByFolder.collect { notes ->
        noteAdapter.submitList(notes)
    }
}
```

Рисунок 2.23 Настройка RecyclerView для обновления списка заметок

```
// Обновляем список папок при изменении данных
viewLifecycleOwner.lifecycleScope.launch {
    mainViewModel.allFolders.collect { folders ->
        folderAdapter.submitList(folders)
    }
}
```


Рисунок 2.24 Настройка RecyclerView для обновления списка папок

Так же отдельно я добавил обновление списка заметок при выборе определенной папки (см. Рисунок 2.25)

```
viewLifecycleOwner.lifecycleScope.launch {
    mainViewModel.selectedFolder.collect { selectedFolder ->
        // Обновление списка заметок для выбранной папки
        mainViewModel.changeListNote(selectedFolder?.id ?: 0)
    }
}
```

Рисунок 2.25 Настройка RecyclerView для обновления списка папок

Навигация и действия: Кнопки для создания новой заметки и перехода к списку папок были добавлены для обеспечения быстрого доступа к этим функциям. Слушатели событий были привязаны к этим кнопкам для выполнения соответствующих действий (см. Рисунок 2.26).

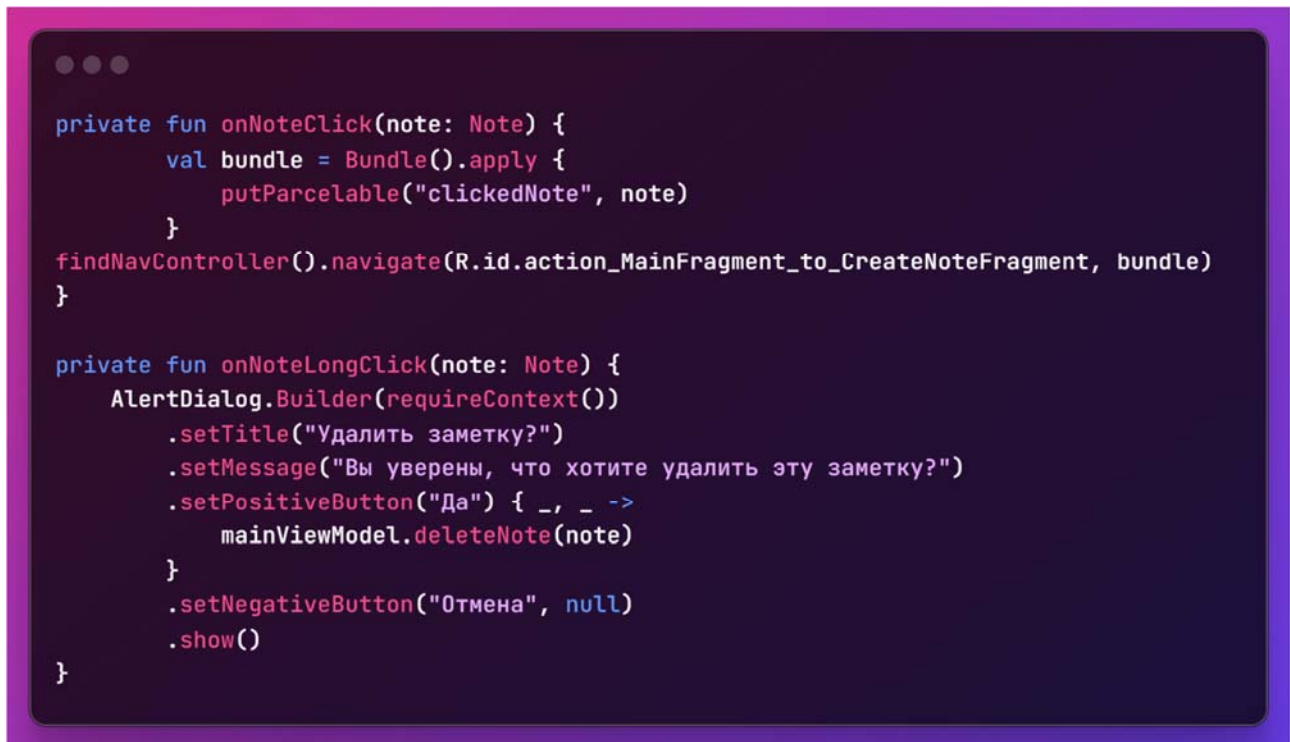
```
binding.buttonCreateNewNote.setOnClickListener {
    findNavController().navigate(R.id.action_MainFragment_to_CreateNoteFragment)
}

binding.buttonNavigationToFolders.setOnClickListener {
    findNavController().navigate(R.id.action_MainFragment_to_ListFoldersFragment)
}
```

Рисунок 2.26 Добавление слушателей кнопок для кнопки перехода к списку папок и кнопки перехода к созданию новой заметки

Обработка событий клика: Для каждой заметки в списке я добавил обработчики кликов, которые позволяли переходить к экрану редактирования

при нажатии на заметку или возможности удалять заметку при долгом нажатии (см. Рисунок 2.27).

A screenshot of a code editor with a dark background and a purple border. The code is written in Kotlin and defines two private functions: `onNoteClick` and `onNoteLongClick`. The `onNoteClick` function creates a `Bundle` with the clicked note and navigates to the `CreateNoteFragment`. The `onNoteLongClick` function shows an `AlertDialog` asking for confirmation to delete the note, with 'Да' (Yes) and 'Отмена' (Cancel) buttons.

```
private fun onNoteClick(note: Note) {  
    val bundle = Bundle().apply {  
        putParcelable("clickedNote", note)  
    }  
    findNavController().navigate(R.id.action_MainFragment_to_CreateNoteFragment, bundle)  
}  
  
private fun onNoteLongClick(note: Note) {  
    AlertDialog.Builder(requireContext())  
        .setTitle("Удалить заметку?")  
        .setMessage("Вы уверены, что хотите удалить эту заметку?")  
        .setPositiveButton("Да") { _, _ ->  
            mainViewModel.deleteNote(note)  
        }  
        .setNegativeButton("Отмена", null)  
        .show()  
}
```

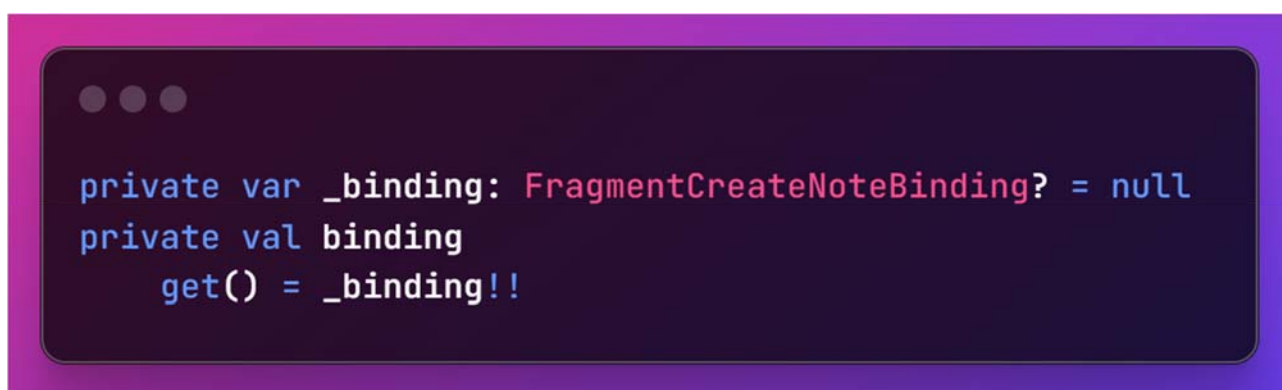
Рисунок 2.27 Добавление действия нажатий на элемент заметки в MainFragment

CreateNoteFragment

Я создал фрагмент CreateNoteFragment, который служит интерфейсом для пользователя при создании новой заметки или редактировании существующей.

Этот фрагмент обеспечивает взаимодействие с пользовательским интерфейсом и логикой приложения.

Привязка данных: Использование класса FragmentCreateNoteBinding позволило мне установить связь между элементами пользовательского интерфейса и данными, что облегчало управление состоянием элементов UI в зависимости от текущего действия пользователя (см. Рисунок 2.28).



```
private var _binding: FragmentCreateNoteBinding? = null
private val binding
    get() = _binding!!
```

Рисунок 2.28 Объявление переменной " FragmentCreateNoteBinding "

Обработка выбранной заметки: В методе onCreateView я реализовал логику, которая определяет, создается новая заметка или редактируется уже существующая. Это достигается путем проверки наличия объекта Note в аргументах фрагмента (см. Рисунок 2.29).

```

val note = arguments?.getParcelable<Note>("clickedNote")
if (note == null) {
    viewModel.createNewNote()
    setCurrentCreationDate()
    lifecycleScope.launch {
        val mainFolder = folderRepository.getMainFolder()
        viewModel.setSelectedFolder(mainFolder!!)
        binding.selectedFolder.text = mainFolder.name
    }
} else {
    viewModel.setCurrentNote(note)
    binding.editTextTitle.setText(note.title)
    binding.editTextContent.setText(note.content)
    binding.textViewDate.text = note.creationDate
    lifecycleScope.launch {
        val currentFolder = folderRepository.getFolderById(note.folderId)
        currentFolder?.let { viewModel.setSelectedFolder(it) }
    }
}

```

Рисунок 2.29 Объявление переменной "FragmentCreateNoteBinding "

Установка выбранной папки: Функция setSelectedFolder отвечает за отображение названия папки, в которой находится заметка. Если заметка не принадлежит ни одной папке, отображается соответствующее сообщение (см. Рисунок 2.30).

```

private fun setSelectedFolder(note: Note?) {
    lifecycleScope.launch {
        val selectedFolderId = note?.folderId ?: 0
        if (selectedFolderId != 0) {
            val selectedFolder = selectedFolderId.let {
                folderRepository.getFolderById(it)
            }
            binding.selectedFolder.text = selectedFolder?.name
        } else {
            binding.selectedFolder.text = "нет папки"
        }
    }
}

```

Рисунок 2.30 Функция " setSelectedFolder()"

Слушатели событий: Я добавил слушатели событий для кнопок “Назад” и “Готово”, которые позволяют пользователю либо вернуться назад без сохранения изменений, либо сохранить заметку и вернуться к основному экрану (см. Рисунок 2.31).

```

private fun setupButtonListeners() {
    binding.buttonBack.setOnClickListener { navigateBack() }
    binding.buttonDone.setOnClickListener { saveNoteAndNavigateBack() }
}

private fun navigateBack() {
    findNavController().navigate(R.id.action_createNoteFragment_to_MainFragment)
}

```

Рисунок 2.31 Слушатели событий для кнопок "Назад" и "Готово"

Сохранение заметки: Метод saveNoteAndNavigateBack инкапсулирует логику сохранения заметки и обновления информации о выбранной папке. Это обеспечивает сохранение всех изменений перед выходом из фрагмента (см. Рисунок 2.32).

```

private fun saveNoteAndNavigateBack() {
    lifecycleScope.launch {
        viewModel.onSaveNote()
        val selectedFolder = viewModel.selectedFolder.value
        if (selectedFolder != null) {
            folderRepository.setSelectedFolder(selectedFolder)
        }
    }
    navigateBack()
}

```

Рисунок 2.32 Метод saveNoteAndNavigateBack()

Работа с текстовыми полями: Для полей ввода заголовка и содержания заметки я использовал TextWatcher, что позволяет отслеживать изменения в тексте и соответственно обновлять состояние модели представления (см. Рисунок 2.33 и 2.34).

```

private fun setEditTextTitleTextChangedListener() {
    binding.editTextTitle.addTextChangedListener(object : TextWatcher {
        override fun beforeTextChanged(
            s: CharSequence,
            start: Int,
            count: Int,
            after: Int
        ) {
            // Ничего не делать
        }

        override fun onTextChanged(s: CharSequence, start: Int, before: Int, count: Int) {
            viewModel.onTitleChanged(s.toString())
        }

        override fun afterTextChanged(s: Editable) {
            // Ничего не делать
        }
    })
}

```

Рисунок 2.33 Установка TextWatcher для поля ввода заголовка

```

private fun setEditTextContentTextChangedListener() {
    binding.editTextContent.addTextChangedListener(object : TextWatcher {
        override fun beforeTextChanged(
            s: CharSequence,
            start: Int,
            count: Int,
            after: Int
        ) {
            // Ничего не делать
        }

        override fun onTextChanged(s: CharSequence, start: Int, before: Int, count: Int) {
            viewModel.onContentChanged(s.toString())
        }

        override fun afterTextChanged(s: Editable) {
            // Ничего не делать
        }
    })
}

```

Рисунок 2.34 Установка TextWatcher для поля ввода тела заметки

Для установки времени создания или редактирования заметки я написал метод `setCurrentCreationDate` (см. Рисунок 2.35)

```

private fun setCurrentCreationDate() {
    lifecycleScope.launch {
        val creationDate = viewModel.getCreationDate()
        binding.textViewDate.text = creationDate
    }
}

```

Рисунок 2.35 Установка TextWatcher для поля ввода тела заметки

ListFoldersFragment

В процессе создания фрагмента ListFoldersFragment для моего приложения “Заметки”, я сосредоточился на предоставлении пользователям удобного способа управления папками. Этот фрагмент должен был позволить пользователям не только просматривать существующие папки, но и создавать новые, а также удалять ненужные.

Инициализация и привязка: с помощью класса FragmentListFoldersBinding, я связал элементы пользовательского интерфейса с логикой приложения, что позволило мне управлять состоянием UI и обрабатывать пользовательские действия (см. Рисунок 2.36).



```
private var _binding: FragmentListFoldersBinding? = null
private val binding
    get() = _binding!!
```

Рисунок 2.36 Объявление переменной " FragmentListFoldersBinding "

Адаптер ListFolderAdapter: Я использовал ListFolderAdapter для отображения списка папок в RecyclerView. Этот адаптер обрабатывает клики по папкам, позволяя пользователям выбирать папку для просмотра заметок или удалять папку при долгом нажатии (см. Рисунок 2.37 и 2.38).

```

private val listFolderAdapter: ListFolderAdapter = ListFolderAdapter(
    onFolderClick = { folder ->
        val sharedPreferences =
            requireActivity().getSharedPreferences("sharedPrefs", Context.MODE_PRIVATE)
        val editor = sharedPreferences.edit()
        editor.putInt("selectedFolderId", folder.id!!)
        editor.apply()
        findNavController().navigate(R.id.action_ListFoldersFragment_to_MainFragment)
    },
    onFolderLongClick = { folder ->
        onFolderLongClick(folder)
    }
)
binding.recyclerViewFolders.adapter = listFolderAdapter

```

Рисунок 2.37 Объявление переменной "ListFolderAdapter "


```

// Обновляем список папок при изменении данных
viewLifecycleOwner.lifecycleScope.launch {
    listFoldersViewModel.allFolders.collect { folders ->
        listFolderAdapter.submitList(folders)
    }
}

```

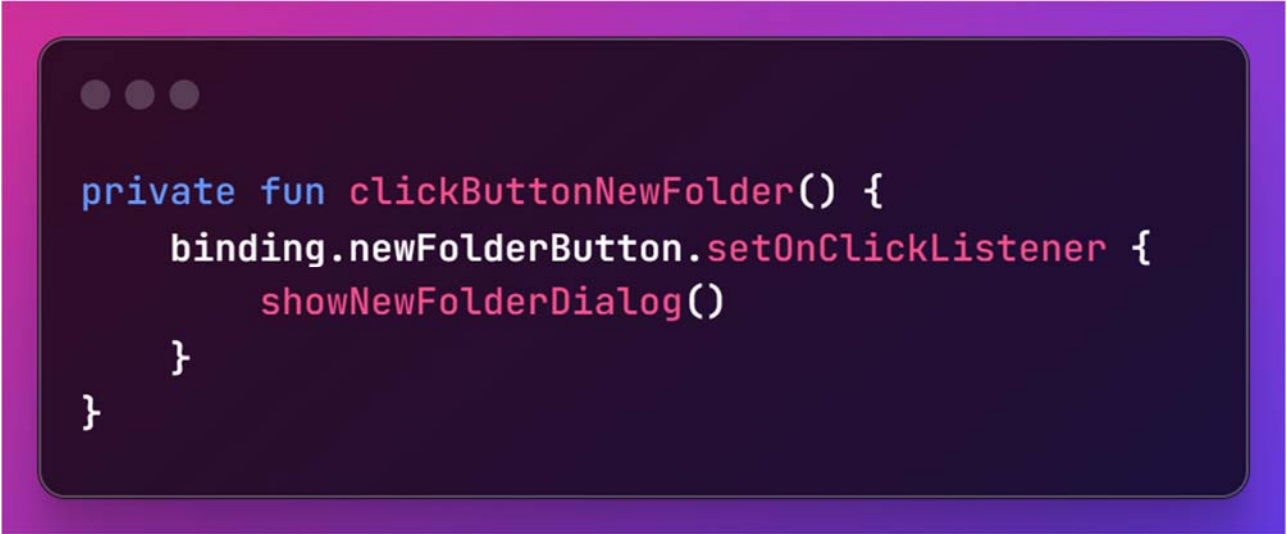
Рисунок 2.38 Обновление списка папок при изменении списка

Обработка событий: В методе `onViewCreated` я настроил слушатели для кнопок “Назад” и “Новая папка”. Кнопка “Назад” позволяет вернуться к основному экрану, а кнопка “Новая папка” открыть диалоговое окно для создания новой папки (см. Рисунки 2.39 и 2.40).



```
private fun clickButtonBack() {  
    binding.buttonBack.setOnClickListener {  
        findNavController().navigate(R.id.action_ListFoldersFragment_to_MainFragment)  
    }  
}
```

Рисунок 2.39 Добавление слушателя кнопки "Назад"



```
private fun clickButtonNewFolder() {  
    binding.newFolderButton.setOnClickListener {  
        showNewFolderDialog()  
    }  
}
```

Рисунок 2.40 Добавление слушателя кнопки "Новая папка"

Создание новой папки: Функция `showNewFolderDialog` отвечает за отображение диалогового окна, в котором пользователь сможет ввести название новой папки. После проверки на уникальность названия, новая папка добавляется в список (см. Рисунок 2.41 и 2.42).

```

private fun showNewFolderDialog() {
    val builder = AlertDialog.Builder(requireContext())
    builder.setTitle("Новая папка")

    val input = EditText(requireContext())
    input.inputType = InputType.TYPE_CLASS_TEXT
    input.filters =
        arrayOf<InputFilter>(InputFilter.LengthFilter(15))
    builder.setView(input)
    builder.setPositiveButton("OK") { dialog, _ ->
        val folderName = input.text.toString()
        if (folderName.isNotEmpty() && !doesFolderExist(folderName)) {
            createNewFolder(folderName)
        } else {
            Toast.makeText(
                requireContext(),
                "Папка с таким названием уже существует",
                Toast.LENGTH_SHORT
            ).show()
        }
        dialog.dismiss()
    }
    builder.setNegativeButton("Отмена") { dialog, _ -> dialog.cancel() }
    builder.show()
}

private fun doesFolderExist(folderName: String): Boolean {
    val allListFolders = listFoldersViewModel.allFolders.value
    return allListFolders.any { it.name == folderName }
}

```

Рисунок 2.41 Функция "showNewFolderDialog"

```

private fun createNewFolder(folderName: String) {
    if (folderName.isBlank()) {
        Toast.makeText(requireContext(), "Имя папки не может быть пустым",
            Toast.LENGTH_SHORT)
            .show()
        return
    }
    val newFolder = Folder(null, folderName, false)
    listFoldersViewModel.insert(newFolder)
}

```

Рисунок 2.42 Функция "createNewFolder"

Удаление папки: Я реализовал функцию onFolderLongClick, которая вызывает диалоговое окно для подтверждения удаления папки. Это обеспечит дополнительный уровень защиты от случайного удаления данных (см. Рисунок 2.43).

```

private fun onFolderLongClick(folder: Folder) {
    AlertDialog.Builder(requireContext())
        .setTitle("Удалить папку?")
        .setMessage("Вы уверены, что хотите удалить эту папку и все заметки в ней?")
        .setPositiveButton("Да") { _, _ ->
            listFoldersViewModel.deleteFolder(folder)
        }
        .setNegativeButton("Отмена", null)
        .show()
}

```

Рисунок 2.43 Функция " onFolderLongClick"

Эти шаги отражают мою стратегию по созданию удобного и функционального интерфейса для работы с заметками в приложении. Каждый аспект был тщательно продуман и реализован с учетом потребностей

пользователя, что в конечном итоге привело к созданию интуитивно понятного и эффективного инструмента для управления заметками.

Для работы этих фрагментов, разумеется, нужны ViewModels, а также классы адаптеры для RecyclerView. Код этих классов представлен в приложениях.

2.3 Тестирование приложения на эмуляторах и реальном устройстве

Тестирование приложения “Заметки” было проведено с использованием различных эмуляторов и реальных устройств, что позволило мне оценить его производительность и устойчивость в разнообразных условиях. Этот этап включал в себя:

1. Использование эмуляторов: Я запустил приложение на эмуляторах с разными версиями Android и разрешениями экрана, чтобы убедиться в его корректной работе на широком спектре устройств.

2. Тестирование на реальных устройствах: Приложение было установлено на несколько физических устройств для проверки его производительности и отзывчивости.

3. Поиск и исправление ошибок: В процессе тестирования были обнаружены и исправлены некоторые ошибки, что повысило стабильность приложения.

4. Оценка пользовательского интерфейса: было проведено тестирование интерфейса на удобство использования, что позволило улучшить пользовательский опыт.

Глава 3. Анализ результатов и предложения по улучшению

3.1 Обзор полученных результатов

Разработка приложения “Заметки” показала, что основные требования к удобству управления заметками были успешно реализованы. Пользователи могут легко создавать, редактировать и удалять заметки, а также организовывать их по папкам. Однако, некоторые функции, такие как мультимедийная поддержка и система напоминаний, не были включены в первоначальную версию приложения из-за ограничений времени и ресурсов.

3.2 Предложения по улучшению приложения

На основе анализа результатов и обратной связи от пользователей, я предлагаю следующие улучшения для будущих версий приложения:

1. Добавление мультимедийной поддержки: Включение возможности прикрепления изображений и аудиофайлов к заметкам.

2. Реализация системы напоминаний: Создание функционала для установки напоминаний по заметкам.

3. Улучшение поиска: Разработка более продвинутой системы поиска по заметкам. Поиск будет организован не только скролированием заметок, распределенным по папкам, но и добавлением поисковой строки, вводя в которую несколько символов, будет автоматическое обновление списка заметок, включающих в себя набор вводимых символов.

4. Оптимизация производительности: Повышение скорости работы приложения и уменьшение времени отклика.

5. Добавление возможности создания "Списка" с удобным интерфейсом для взаимодействия пользователей с ним.

Заключение

В заключении моего исследования и разработки приложения “ GB Notes” можно сделать вывод, что основная цель была достигнута: создано функциональное приложение для управления заметками. Задачи выполнены, хотя и с некоторыми компромиссами из-за ограниченных ресурсов. Результаты исследования имеют практическую значимость, так как приложение может быть использовано для повышения продуктивности и организации информации. В будущем планируется дальнейшее совершенствование приложения, включая реализацию предложенных улучшений и добавление новых функций. Достижение ряда поставленных целей и выполнение задач подтверждают успешность проекта и открывают путь для его дальнейшего развития.

Список использованной литературы

1. Интернет ресурс - <https://gb.ru/> "Образовательная платформа GeekBrains"
2. Интернет ресурс - <https://m2.material.io/> "Material Design 2"
3. Интернет ресурс - <https://www.android.com/> "Android"
4. Интернет ресурс - <https://stackoverflow.com/> "Stackoverflow"
5. Интернет ресурс - <https://www.youtube.com/> "YouTube"

Приложение 1

(обязательное)

Код Класса MainViewModel

```
class MainViewModel @Inject constructor(
    private val noteRepository: NoteRepository,
    private val folderRepository: FolderRepository
) : ViewModel() {
    private val _allNotesByFolder = MutableStateFlow<List<Note>>(listOf())
    val allNotesByFolder = _allNotesByFolder

    private val _allFolders = MutableStateFlow<List<Folder>>(listOf())
    val allFolders = _allFolders

    private val _selectedFolder = MutableStateFlow<Folder?>(null)
    val selectedFolder: StateFlow<Folder?> = _selectedFolder

    init {
        viewModelScope.launch {
            _allFolders.value = folderRepository.getAllFolders()
            createMainFolder()
        }
    }

    fun deleteNote(note: Note) = viewModelScope.launch {
        noteRepository.delete(note)
        _allNotesByFolder.value = noteRepository.getAllNotes()
    }

    fun changeListNote(folderId: Int) {
        viewModelScope.launch {
            val currentFolder = folderRepository.getFolderById(folderId)
            if (currentFolder != null) {
                folderRepository.setSelectedFolder(currentFolder)
                _selectedFolder.value = currentFolder
                val notesInFolder = if (currentFolder.id == 1)
                    noteRepository.getAllNotes()
                else
                    noteRepository.getNotesBySelectedFolder(folderId)
                _allNotesByFolder.value = notesInFolder
            }
        }
    }

    // Функция для вставки новой папки
    private fun insertFolder(folder: Folder) = viewModelScope.launch {
        folderRepository.insert(folder)
        updateFolders()
    }

    private fun createMainFolder() {
        viewModelScope.launch {
            val mainFolderName = "Все"
            val firstFolder = folderRepository.getFolderById(1)
            if (firstFolder == null) {
                val mainFolder = Folder(id = null, name = mainFolderName, isSelected =
true)
                insertFolder(mainFolder)
            }
        }
    }

    // Функция для обновления
    fun updateFolders() {
        viewModelScope.launch {
            allFolders.value = folderRepository.getAllFolders()
        }
    }
}
```


Приложение 2 (обязательное)

Код Класса CreateNoteViewModel

```
class CreateNoteViewModel @Inject constructor(
    private val noteRepository: NoteRepository,
    private val folderRepository: FolderRepository
) : ViewModel() {
    private val _note = MutableStateFlow(Note(0, 0, "", "", ""))
    val note: StateFlow<Note> = _note

    private val _currentNote = MutableStateFlow<Note?>(null)
    val currentNote: StateFlow<Note?> = _currentNote

    private val _isNoteModified = MutableStateFlow(false)
    val isNoteModified: StateFlow<Boolean> = _isNoteModified

    private val _allFolders = MutableStateFlow<List<Folder>>(listOf())
    val allFolders: StateFlow<List<Folder>> = _allFolders

    private val _allFolderNames = MutableStateFlow<List<String>>(emptyList())
    val allFolderNames: StateFlow<List<String>> = _allFolderNames

    private val _selectedFolder = MutableStateFlow<Folder?>(null)
    val selectedFolder: StateFlow<Folder?> = _selectedFolder

    private val _isNewNote = MutableStateFlow(true)
    val isNewNote: StateFlow<Boolean> = _isNewNote

    init {
        viewModelScope.launch {
            val folderList = folderRepository.getAllFolders()
            val folderNames = folderList.map { it.name }
            _allFolderNames.value = folderNames
            _allFolders.value = folderList
        }
    }

    fun onTitleChanged(newTitle: String) {
        _note.value = _note.value.copy(title = newTitle)
        _isNoteModified.value = true
    }

    fun onContentChanged(newContent: String) {
        _note.value = _note.value.copy(content = newContent)
        _isNoteModified.value = true
    }

    fun getCreationDate(): String {
        val currentDate =
            LocalDateTime.now().format(DateTimeFormatter.ofPattern("dd.MM.yyyy"))
        val currentTime = LocalTime.now().format(DateTimeFormatter.ofPattern("HH:mm "))
        return currentDate + currentTime
    }

    fun setCurrentNote(note: Note) {
        _currentNote.value = note
        _note.value = note
        _isNewNote.value = false
    }

    fun setSelectedFolder(folder: Folder) {
        _selectedFolder.value = folder
    }

    fun createNewNote() {
```

```

        val creationDate = getCreationDate()
        _note.value = Note(0, 0, "", "", creationDate)
        _isNewNote.value = true
    }

    fun onSaveNote() {
        viewModelScope.launch {
            val currentNote = _note.value

            var selectedFolder = _selectedFolder.value

            if (selectedFolder == null) {
                selectedFolder = folderRepository.getFolderById(1)
                _selectedFolder.value = selectedFolder
            }

            val noteWithFolderId = currentNote.copy(folderId = selectedFolder?.id!!)
            val noteWithCreationDate =
                if (_currentNote.value != null && _isNoteModified.value) {
                    // Если текущая заметка существует и была изменена, обновляем дату
                    val creationDate = getCreationDate()
                    noteWithFolderId.copy(creationDate = creationDate)
                } else {
                    noteWithFolderId
                }

            if (_isNewNote.value) {
                // Если создается новая заметка, вставляем ее
                noteRepository.insert(noteWithCreationDate)
                folderRepository.update(selectedFolder)
            } else {
                // Если редактируется существующая заметка, обновляем ее
                noteRepository.update(noteWithCreationDate)
            }
        }
    }

    fun onFolderSelected(position: Int) {
        _selectedFolder.value = _allFolders.value[position]
        viewModelScope.launch {
            val selectedFolder = _selectedFolder.value
            if (selectedFolder != null) {
                folderRepository.setSelectedFolder(selectedFolder)
            }
        }
    }
}

```

Приложение 3 (обязательное)

Код Класса ListFoldersViewModel

```
class ListFoldersViewModel @Inject constructor(
    private val folderRepository: FolderRepository
) : ViewModel() {
    private val _allFolders = MutableStateFlow<List<Folder>>>(listOf())
    val allFolders = _allFolders

    init {
        viewModelScope.launch {
            _allFolders.value = folderRepository.getAllFolders()
        }
    }

    // Функция для вставки новой папки
    fun insert(folder: Folder) = viewModelScope.launch {
        folderRepository.insert(folder)
        updateFolders()
    }

    // Функция для обновления
    private fun updateFolders() {
        viewModelScope.launch {
            allFolders.value = folderRepository.getAllFolders()
        }
    }

    fun deleteFolder(folder: Folder) = viewModelScope.launch {
        folderRepository.deleteFolder(folder)
        _allFolders.value = folderRepository.getAllFolders()
    }
}
```

Приложение 4 (обязательное)

Код Класса **MainFolderAdapter**

```
class FolderAdapter(  
    private val onChangeShowNote: (folderId: Int) -> Unit,  
) : ListAdapter<Folder,  
FolderAdapter.MainFolderViewHolder>(FolderDiffUtilCallback()) {  
  
    override fun onCreateViewHolder(parent: ViewGroup, viewType:  
Int): MainFolderViewHolder {  
        return MainFolderViewHolder(  
            ItemFolderToMainFragementBinding.inflate(  
                LayoutInflater.from(parent.context),  
                parent,  
                false  
            )  
        )  
    }  
  
    override fun onBindViewHolder(holder: MainFolderViewHolder,  
position: Int) {  
        val currentFolder = getItem(position)  
        holder.bind(currentFolder)  
    }  
  
    inner class MainFolderViewHolder(private val binding:  
ItemFolderToMainFragementBinding) :  
        RecyclerView.ViewHolder(binding.root) {  
        fun bind(folder: Folder) {  
            binding.buttonFolderInMainRecyclerView.text = folder.name  
            binding.buttonFolderInMainRecyclerView.setOnClickListener  
{  
                folder.id?.let { it1 -> onChangeShowNote(it1) }  
            }  
        }  
    }  
}  
  
class FolderDiffUtilCallback : DiffUtil.ItemCallback<Folder>() {  
    override fun areItemsTheSame(oldItem: Folder, newItem: Folder) =  
        oldItem.id == newItem.id  
  
    override fun areContentsTheSame(oldItem: Folder, newItem: Folder)  
=  
        oldItem == newItem  
}
```

Приложение 5 (обязательное)

Код Класса ListFolderAdapter

```
class ListFolderAdapter(  
    private val onFolderClick: (Folder) -> Unit,  
    private val onFolderLongClick: (Folder) -> Unit  
) : ListAdapter<Folder,  
ListFolderViewHolder>(ListFolderDiffUtilCallback()) {  
  
    override fun onCreateViewHolder(parent: ViewGroup,  
viewType: Int): ListFolderViewHolder {  
        val binding = ItemFolderToListBinding.inflate(  
            LayoutInflater.from(parent.context),  
            parent,  
            false  
        )  
        return ListFolderViewHolder(binding)  
    }  
  
    override fun onBindViewHolder(holder:  
ListFolderViewHolder, position: Int) {  
        val currentFolder = getItem(position)  
        holder.bind(currentFolder)  
        holder.itemView.setOnClickListener {  
onFolderClick(currentFolder) }  
        holder.itemView.setOnLongClickListener {  
            onFolderLongClick(currentFolder)  
            true  
        }  
    }  
}  
  
class ListFolderDiffUtilCallback :  
DiffUtil.ItemCallback<Folder>() {  
    override fun areItemsTheSame(oldItem: Folder, newItem:  
Folder) =  
        oldItem.id == newItem.id  
  
    override fun areContentsTheSame(oldItem: Folder, newItem:  
Folder) =  
        oldItem == newItem  
}  
  
class ListFolderViewHolder(  
    private val binding: ItemFolderToListBinding,  
) : RecyclerView.ViewHolder(binding.root) {  
    fun bind(folder: Folder) {
```

```
binding.apply {  
    folderName.text = folder.name  
    checkmark.isVisible = folder.isSelected  
}  
}
```

Приложение 6 (обязательное)

Код Класса **NoteAdapter**

```
class NoteAdapter(  
    private val clickNote: (note: Note) -> Unit,  
    private val longClickNote: (note: Note) -> Unit  
) : ListAdapter<Note, NoteAdapter.NoteViewHolder>(  
    DiffUtilCallback()  
) {  
  
    override fun onCreateViewHolder(parent: ViewGroup, viewType:  
Int): NoteViewHolder {  
        val binding = ItemNoteToMainFragmentBinding.inflate(  
            LayoutInflater.from(parent.context),  
            parent,  
            false  
        )  
        return NoteViewHolder(binding)  
    }  
  
    override fun onBindViewHolder(holder: NoteViewHolder,  
position: Int) {  
        val currentNote = getItem(position)  
        holder.bind(currentNote)  
    }  
  
    inner class NoteViewHolder(  
        private val binding: ItemNoteToMainFragmentBinding,  
    ) :  
        RecyclerView.ViewHolder(binding.root) {  
        fun bind(note: Note) {  
            binding.apply {  
                noteTitle.text = note.title  
                noteContent.text = note.content  
                noteDate.text = note.creationDate.toString()  
                root.setOnClickListener {  
                    clickNote(note)  
                }  
                root.setOnLongClickListener {  
                    longClickNote(note)  
                    true  
                }  
            }  
        }  
    }  
}  
  
class DiffUtilCallback : DiffUtil.ItemCallback<Note>() {  
    override fun areItemsTheSame(oldItem: Note, newItem: Note) =  
        oldItem.id == newItem.id  
}
```

```
override fun areContentsTheSame(oldItem: Note, newItem: Note)  
=  
    oldItem == newItem  
}
```