Geekbrains

Android приложение для журналирования доходов и расходов

IT специалист

Программист мобильных устройств. Цифровые профессии

Садретдинов Р.Х.

Москва

2024

Оглавление

Глава 1

Введение

Развитие приложений для мобильных телефонов продолжается динамично, и за последние несколько лет наблюдаются несколько ключевых тенденций.

Искусственный интеллект и машинное обучение: использование искусственного интеллекта (ИИ) и технологий машинного обучения (МО) становится все более распространенным в мобильных приложениях. Это включает в себя функции распознавания речи, компьютерного зрения, персонализированных рекомендаций и другие сценарии, которые улучшают опыт пользователей. Говорят (я сам не видел, читал про это в Интернете) последние флагманы от Samsung включают в себя не только ИИ который был ранее и был в основном ориентирован на обработку фотографий, но и "синхронный переводчик", поиск по картинам, основанный на использовании ИИ, интеллектуальный помощник в

составлении заметок и многое-многое другое.

Расширенная реальность (AR) и виртуальная реальность (VR): приложения, использующие технологии AR и VR, тоже становятся всё более популярными день ото дня. Они включают в себя игры, образовательные приложения, рекламные кампании и даже инструменты для дополненной реальности в различных областях.

Развлечения и социализация: мобильные приложения широко используются для развлечения и социального взаимодействия. Игры, социальные сети, стриминговые сервисы, мессенджеры — всё это важные аспекты развлечений и общения в современном мире.

Интернет вещей (IoT): с ростом числа подключенных устройств приложения для мобильных телефонов начинают интегрировать функциональность для управления и мониторинга умных устройств в доме, автомобиле и других сферах жизни.

Финансовые технологии (финтех): развитие мобильных приложений в сфере финансовых технологий активно продолжается. Электронные кошельки мобильные банки, инвестиционные платформы и сервисы для цифровых платежей становятся более удобными и функциональными.

Блокчейн и криптовалюты: некоторые приложения начинают интегрировать технологии блокчейна и криптовалюты для обеспечения безопасности, прозрачности и децентрализации. Это может касаться цифровых кошельков, децентрализованных приложений (DApps) и других блокчейн-решений.

Забота о здоровье и благополучии: приложения для мобильных устройств, связанные с фитнесом, медицинским мониторингом, психическим здоровьем и заботой о благополучии, становятся более популярными. Они предоставляют пользователям инструменты для отслеживания физической активности, сна, питания и других аспектов здорового образа жизни.

Медицинская помощь и образование: мобильные приложения играют важную роль в области медицины и образования. Они предоставляют доступ к медицинской информации, образовательным ресурсам, тренировочным программам и многому другому, улучшая уровень заботы о здоровье и возможности обучения. Я сам пользуюсь приложением ЕМИАС для записи к врачу, просмотра результатов медицинских обследований, всегда есть возможность посмотреть свою медицинскую карту не посещая поликлинику, а прямо из дома...

Удобство и доступность: мобильные приложения обеспечива-

ют удобный и моментальный доступ к информации и сервисам прямо с мобильного устройства. Это позволяет пользователям быстро находить необходимую информацию, выполнять задачи и взаимодействовать с сервисами в любом месте и в любое время.

Персонализация и адаптация: многие приложения предоставляют персонализированный опыт, учитывая предпочтения и поведение пользователя. Это позволяет создавать уникальные и релевантные интерфейсы, предоставлять рекомендации и оптимизировать взаимодействие с приложением.

Улучшение производительности: многие и профессиональные приложения предназначены для улучшения производительности и эффективности. Они предоставляют инструменты для работы с документами, управления задачами, планирования и другие функции, которые помогают людям быть более продуктивными.

Электронная коммерция: приложения для мобильных устройств стали неотъемлемой частью электронной коммерции. Они позволяют пользователям легко совершать покупки, отслеживать статус заказов, получать уведомления о скидках и акциях, делать цифровые платежи и многое другое.

Инновации и новые технологии: развитие мобильных приложе-

ний стимулирует внедрение новых технологий, таких как искусственный интеллект, расширенная реальность, блокчейн и другие. Это способствует развитию индустрии и обогащению пользовательского опыта.

Безопасность и контроль: многие мобильные приложения обеспечивают возможность контроля за личной информацией, финансами, безопасностью устройства. Это важно для пользователей, когда речь идет о сохранности данных и защите личности.

Глава 2

Теоретические аспекты приложения

2.1 Рассуждения...

Написание мобильного приложения для учета доходов и расходов включает в себя несколько ключевых шагов. Задумка у меня, безусловно, "титаническая". Но в связи с:

1. недостаток времени
2. это моя *первая в жизни* программа для Android
3. к диплому надо было сделать хоть что-то работоспособное чтобы продемонстрировать что курс *"Программист мобильных устройств. Цифровые профессии"* я посещал не зря.

Назвать своё приложение я решил просто: **"Bookkeeper"**, что

в переводе значит **"Счетовод"**.

Цель приложения: **"Bookkeeper"**предназначен для удобного учёта моих личных финансов, избавляя меня от необходимости каждый месяц записывать свои доходы и расходы на бумажку, что я делал ежемесячно начиная с августа 2020 года. Этих бумажек у меня скопилась уже гора, к тому же я иногда ошибался, складывая и вычитая все эти цифры вручную, и это было обиднор потому что приходилось всё пересчитывать с самого начала месяца.

Основные функции приложения таковы:

Так как это приложение я сделал для себя лично (ну пока ещё не доделал как хотелось, но для представления в качестве дипломной работы, надеюсь, что сделал), не планирую его никуда распространять и пользоваться буду сам, то никакая регистрация, профиль пользователя, вход по паролю и идентификация личности по биометрии мне в нём не нужны.

А вот что реально нужно:

1. Общий баланс и текущее финансовое положение.
2. Добавление нового дохода с указанием источника, суммы и даты.

3. Добавление нового расхода с указанием категории, суммы и даты.

4. Возможность создания пользовательских категорий расходов (на моих бумажках подобное я заносил в пункт "Непредвиденные расходы".

5. Установка месячных или еженедельных бюджетов.

6. Уведомления что мой бюджет почему-то приближается к завершению.

7. Напоминания о предстоящих счетах, платежах и других финансовых обязательствах (на моих бумажках подобное я заносил в пункт "Обязательные расходы". .

8. **Обязательно** синхронизация с какой-нибудь внешней базой данных, хоть в облаке, хоть даже на локальном компьютере — телефон вполне можно и потерять где-нибудь, обидно будет если вместе с ним потеряется и вся база движения моих личных финансов, а на локальном компьютере или тем более где-нибудь в облаке эта база будет доступна практически всегда.

Первые три пункта я сделал. Остальные обязательно сделаю.

Технологический стек:

- Язык программирования: пока что Java для Android. Для iOS я ничего напрограммировать не могу по причине физического отсутствия у меня iPhone, соответственно мне приложение для iOS будет полностью бесполезно и по причине физического отсутствия у меня macbook, соответственно мне просто не на чём программировать — XCode запускается лишь на macOS, а у меня отсутствует возможность его запустить. В то время как Android Studio запускается под всеми существующими OS, да и телефон у меня Android.

  Конечно, в силу современного тренда для программирования под Android надо бы использовать Kotlin вместо Java, но на курсе *"Программист мобильных устройств. Цифровые профессии"*нам не преподавали Kotlin. Более того, Android нам тоже не преподавали, но я попросил техподдержку Geekbrains и мне открыли старые курсы по Android, факультативно, которые когда-то у Geekbrains имелись. Так как курсы по нынешним временам уже довольно старые, то ни о каком Kotlin там речи не было. Я эти курсы пересмотрел, чему-то научился и в результате вот, представляю свой дипломный проект. Хоть и не iOS, а Android, но моей специа-

лизации, по которой я обучался на курсах Geekbrains, *"Программист мобильных устройств"*, вполне соответствует.

- Использование локальной базы данных на телефоне для хранения финансовых данных.

Я попытался сделать приложение согласно всем канонам, которым нас обучали на модуле "Архитектура ПО". И считаю что у меня вполне получилось сделать приложение удовлетворяющее паттерну **MVC**.

2.2 Среда разработки

На сегодняшний день для написания программ под Android большинство использует Android Studio. Можно, конечно, использовать и IntelliJ IDEA, и NetBeans, и Eclipse, и даже последние версии Visual Studio от Microsoft и Embarcadero Delphi позволяют создавать приложения для Android, но я остановился именно на Android Studio.

Была использована самая последняя версия "Android Studio Hedgehog | 2023.1.1 Patch 2", разработка велась на компьютере под управлением Fedora Linux 39.

2.3 База данных, используемая в приложении

Мобильные телефоны, как и многие другие электронные устройства, нуждаются в каком-то хранилище данных. Причём это хранилище должно быть постоянным (persistence), чтобы данные не терялись при выключении телефона. К таким постоянно требующемся данным, которые терять очень нежелательно, можно отнести контакты, сообщения SMS и MMS, пользовательские заметки.

Конечно, можно было бы для каждого приложения, которое

нуждается в подобной функциональности, создавать своё хранилище, но это довольно непрактично. К счастью, в телефонах уже существуют подобные встроенные базы данных и ничего собственного придумывать не надо.

К примеру, в телефонах фирмы Apple для постоянного, то есть не критичного к выключению телефона, хранения данных используется Core Data[1], а в телефонах Android используется SQLite[2].

Так как тема данного дипломного проекта про приложение для телефонов на базе Android, то на Core Data, которая используется в телефонах фирмы Apple, я останавливаться не буду.

SQLite это легковесная встроенная реляционная база данных. Основные моменты, относящиеся к SQLite, которые хотелось бы отметить:

- **Тип базы данных:** SQLite является встроенной базой данных, предназначенной для мобильных устройств. Она хранится в виде файла на устройстве.

- **Легкость:** SQLite отличается от больших реляционных СУБД, таких как MySQL или PostgreSQL, своей легкостью и минимализмом, что делает её идеальным выбором для ограниченных по ресурсам мобильных устройств.

- **API для работы:** Android предоставляет API для взаимодействия с SQLite из приложений на Java или Kotlin.
- **SQLiteOpenHelper:** Для управления базой данных и ее версиями в Android используется класс SQLiteOpenHelper.
- **Таблицы:** Данные хранятся в таблицах, каждая из которых имеет уникальное имя.
- **Строки и столбцы:** Данные в таблицах представлены строками и столбцами.
- **SQL-запросы:** Используются для вставки, обновления, выборки и удаления данных.
- **CRUD-операции:** Создание (Create), чтение (Read), обновление (Update) и удаление (Delete) данных выполняются с использованием SQL-запросов.
- **Транзакции:** Позволяют группировать несколько операций в одну атомарную единицу работы (в данном дипломном проекте транзакции не используются по причине того, что база состоит из всего одной единственной таблицы и поэтому группировать тут просто напросто нечего).

Поэтому для своего приложения я выбрал именно SQLite.

К небольшим недостаткам SQLite можно отнести немного сложное и запутанное программирование работы с этой СУБД. Но, как я уже заметил выше, Google предоставил API, которое в разы облегчает процесс использования этой базы данных в своих приложениях. Этот API называется Android Room.

2.4 Android Room

Android Room — это библиотека для работы с базой данных в приложениях Android. Она предоставляет уровень абстракции над SQLite, облегчая работу с базой данных и улучшая производительность приложения. Вот несколько основных преимуществ Android Room по сравнению с использованием "голого"и "сырого"SQLite:

- **Удобство использования:** Room предоставляет простой и удобный API для работы с базой данных, что делает код более читаемым и поддерживаемым.
- **Абстракция от SQL:** Room предоставляет абстракцию от непосредственного написания SQL-запросов. Он использует аннотации для определения сущностей базы данных и их связей, что упрощает процесс разработки.
- **Компиляционная проверка запросов:** Room проводит компиляционную проверку SQL-запросов во время сборки проекта, что помогает обнаруживать ошибки на этапе компиляции, а не во время выполнения приложения.
- **Поддержка LiveData:** Room интегрируется хорошо с архитектурой компонентов Android, такой как LiveData, что

упрощает реактивное программирование. Также есть поддержка RxJava для тех, кто предпочитает использовать RxJava в своих проектах.

- **Управление версиями и миграции:** Room предоставляет механизм управления версиями базы данных и автоматическое выполнение миграций при изменениях схемы базы данных.

- **Поддержка аннотаций для SQL-запросов:** Room позволяет использовать аннотации для определения SQL-запросов, что уменьшает шанс допущения ошибок при их написании.

В целом, использование Android Room упрощает разработку приложений с использованием баз данных SQLite, предоставляя более высокий уровень абстракции и инструментов для повышения производительности и надежности кода.

2.5 Почему LiveData?

Как и при программировании для iOS при программировании для Android необходимо чтобы UI (User Interface, пользовательский интерфейс) приложения был как мжно более "отзывчивым"и не заставлял долго ждать ответа на действия пользователя. К тому же в Android существует так называемый ANR (Application Not Responding) — система Android сама следит за временем ответной реакции приложения на действия пользователя и если это время превышает определённое значение, то выводится окно, в котором сообщается что данное приложение не отвечает и предлагается либо его без вопросов закрыть, либо ещё подождать опять определённое время. Такое происходит как правило если пытаться выполнять довольно тяжеловесные и долгие задачи в основном потоке приложения. А основной поток приложения предназначен как раз только для интерфейса с пользователем. Поэтому все операции, которые требуют достаточного количества времени на реакцию, как то операции ввода-вывода, операции обращения к интернету, какие-нибудь долгие подсчёты и подобное должны выполняться в отдельном потоке, который занимается только этим и никак не может оказать влияние на быстродействие взаимодействия с пользователем. К примеру, в

iOS для взаимодействия с интернетом создаётся объект так называемой "сессии", URLSession, который и выполняет взаимодействие с сетью, не оказывая никакого влияния на основной поток, который занят лишь общением с пользователем.

Android Room просто откажется работать, если определит что его вызывают не из вспомогательного потока, а из основного (к слову, такое поведение Android Room можно отключить, передав определённый параметр при создании объекта базы данных, а именно `allowMainThreadQueries()`, но делать это крайне не рекомендуется потому что это запросто может привести к ANR).

Поэтому запросы к Android Room выполняются асинхронно, в другом потоке. Как правило для этого используется класс `Executor` и его метод `execute()`.

А вот для слежения за тем, что асинхронный метод Android Room завершил свою работу как раз и применяется LiveData.

LiveData — это компонент архитектуры компонентов Android, предоставляющий наблюдаемые данные для построения быстрого и удобного интерфейса, не заставляющего пользователя долго ожидать ответа на свои действия. Он предназначен для упрощения обработки данных в реальном времени, таких как данные из базы данных, сетевых запросов или других источников.

Все объекты LiveData обладают методом observer (наблюдатель), посредством которого и получают сигнал о том, что асинхронный метод Android Room завершился.

Преимущества использования LiveData по сравнению с обычным, синхронным подходом:

- **Автоматическое управление жизненным циклом:** LiveData автоматически учитывает жизненный цикл компонентов Android, таких как Activity или Fragment. Это предотвращает утечки памяти и ошибки, связанные с доступом к устаревшим данным.

- **Обновление пользовательского интерфейса:** LiveData позволяет автоматически обновлять пользовательский интерфейс, когда данные изменяются. Компоненты, подписанные на LiveData, будут получать уведомления об изменениях и обновлять себя соответствующим образом.

- **Интеграция с архитектурой компонентов Android:** LiveData является частью архитектурных компонентов Android и хорошо интегрируется с другими компонентами. Это обеспечивает согласованный и эффективный способ разработки приложений.

Теперь, когда речь идет о сочетании LiveData с Android Room:

- **Обновление UI при изменении данных в базе данных:** Использование LiveData в связке с Android Room позволяет легко обновлять пользовательский интерфейс при изменении данных в базе данных. Результаты запросов к базе данных, возвращаемые Room, могут быть обернуты в LiveData. Это позволяет автоматически обновлять UI, когда данные в базе данных меняются.

- **Автоматическое управление жизненным циклом:** LiveData, используемый в связке с Room, обеспечивает автоматическое управление жизненным циклом, что предотвращает утечки памяти и проблемы с обновлением UI в неподходящих моментах.

- **Легкость интеграции с Android Room:** Когда LiveData используется в связке с Room, это позволяет создавать эффективные и чистые архитектуры приложений, разделяя логику обработки данных и представления.

Применение LiveData совместно с Android Room улучшает структуру и производительность кода, обеспечивая тем самым эффективное управление данными и их отображение в пользова-

тельском интерфейсе.

Глава 3

Практическое выполнение приложения

3.1 Структура приложения и его старт

Приложение **"Bookkeeper"**можно разделить на три слоя — Model, которая содержит саму базу данных, View ("представление"на самом деле не одно, их много, на каждый экран своё) и Controller, который по большей части сам ничего не делает, но содержит в себе большинство функций и методов, необходимых практически каждому представлению. Так же в Controller содержатся все методы доступа к Model.

Приложение "одноактивное", то есть в нём содержится только одна Activity, а все экраны реализованы в виде фрагментов. В некоторых приложениях для Android каждый раз создаётся но-

вая ативность для каждого нового экрана, но я решил что пусть активность будет одна, и именн она пусть будет контроллером, содержащим в себе всё необходимое для работы с моделью, то есть репозиторием и базой данных и представлениями. А за каждый экран уже пусть отвечает конкретное представление этого экрана.

id	int
name	String
flow_date	Date
amount	int

Таблица 3.1: money_flows

3.2 Детальное описание базы данных приложения

База данных, используемая в **"Bookkeeper"**, простейшая, состоит всего из одной таблицы (Таблица 3.1)

В поле **id** типа `int` хранится уникальный идентификационный номер записи. Номера не повторяются, они объявлены как `Primary key` и автогенерируются самим SQLite.3.2

В поле **name** типа `String` записано место, где была произведена данная финансовая операция (покупка в магазине, оплата счёта, получение долга и т.д.)

В поле **flow_date** типа `Date` записана дата и время операции. Так как в SQLite отсутствует тип для хранения даты (в SQLite весьма ограниченный набор типов, имеются только `NULL, INTEGER, REAL, TEXT` и `BLOB`), то был написан класс-конвертер для преобразования стандартного типа `Date` языка Java в тип `INTEGER` для SQLite и наоборот3.2, благо Android Room

позволяет делать такую конвертацию на лету.

В поле **amount** типа `int` хранится сумма проведённой операции. Если сумма отрицательная, то значит было списание, то есть расход. Если положительная, то значит было пополнение. Это довольно удобно потому что по знаку поля **amount** можно сразу же определить тип операции, кроме того простым сложением можно посчитать и общий приход, и общий расход за требуемый промежуток времени.

Я не стал хранить сумму в виде `REAL`, всё равно никакие проценты мне высчитывать не надо, поэтому храню сумму в копейках. Просто при вводе умножаю введённое число на 100 чтобы порлучить целое число в копейках, а при выводе сначала вывожу результат от целочисленного деления числа в поле **amount** на 100, после десятичную точку, после результат от остатка деления числа в поле **amount** на 100, получается как раз сумма в рублях и копейках.

Для создания/хранения данной таблицы согласно документации на Android Room был создан вот такой класс:

```
@Entity(tableName = "money_flows")
    public class MoneyFlow {
    @ColumnInfo(name = "id")
    @PrimaryKey(autoGenerate = true)
    private int mId;
```

```
@ColumnInfo(name = "name")
private String mName;

@ColumnInfo(name = "flow_date")
@NonNull
@TypeConverters({DateConverters.class})
private Date mFlowDate;

@ColumnInfo(name = "amount")
private int mAmount;

public int getId() {
    return mId;
}

public void setId(int id) {
    mId = id;
}

public String getName() {
    return mName;
}

public void setName(String name) {
    mName = name;
}

public Date getFlowDate() {
    return mFlowDate;
}

public void setFlowDate(Date flowDate) {
    mFlowDate = flowDate;
```

```
    }

    public int getAmount() {
        return mAmount;
    }

    public void setAmount(int amount) {
        mAmount = amount;
    }
}
    \label{src:money_flows_class}
```

Android Room требует чтобы поля в классе, представляющем таблицу в SQLite были или `public`, или имели соответствующие геттеры и сеттеры. Я сделал геттеры и сеттеры.

Аннотация `@Entity` указывает Android Room как будет называться представление этого класса в базе данных SQLite — для языка Java это класс с именем `MoneyFlow`, а для SQLite жто таблица с именем `money\_flows`

Аннотация `@@ColumnInfo` указывает Android Room как будет называться столбец в этой таблице, тоже потому что в Java название одно, а в SQLite другое и надо чтобы Android Room знал чего чему соответствует.

Аннотация `@PrimaryKey` указывает что это поле как раз и будет первичным ключом таблицы. И то что указано `autoGenerate = true` говорит о том, что это поле будет автоинкрементным в базе и SQLite сам будет отвечать за его заполнение.

Аннотация `TypeConverters` как раз говорит что для передачи значения этого поля между Java и SQlite необходимо использовать класс-конкернер, в данном случае класс-конвертер с именем `DateConverters`

Класс этот простейший:

```
public class DateConverters {
```

```
    @TypeConverter
    public static Date fromTimestamp(Long value) {
        return new Date(value);
    }

    @TypeConverter
    public static Long dateToTimestamp(Date date) {
        if (date == null) {
            return 0L;
        }
        Calendar calendar = Calendar.getInstance();
        calendar.setTime(date);
        calendar.set(Calendar.MILLISECOND, 0);
        calendar.set(Calendar.SECOND, 0);
        return calendar.getTimeInMillis();
    }
}
```

Вся его работа заключается в том, чтобы при передаче объекта типа `Date` в SQLite преобразовать его в unixtime (миллисекунды с начала эпохи, то есть сколько прошло миллисекунд с 1 января 1970 года). Ну и при передаче значения из SQlite в Java преобразовать миллисекунды в объект типа `Date`.

3.3 Непосредственно работа приложения

После запуска приложение первым делом инициализирует репозиторий:

```
public class BookkeeperApplication extends Application {
    Override
    public void onCreate() {
        super.onCreate();

        BookkeeperRepository.initRepository(this,
        this.getResources().getString(R.string.database_name));
    }
}
```

Можно с натяжкой назвать класс `BookkeeperApplication` синглтоном, хотя на самом деле он таковым не является, а уже OS Android заботится о том, чтобы во время жизни приложения существовал один и только один класс вида `Application` (прям как класс `AppDelegate` в Swift для iOS).

Затем OS Android вызывает ту активность, которая в файле манифеста помечена как `android.intent.action.MAIN`, это как раз `BookkeeperActivity`. У неё вызывается метод `onCreate()` (ну ей-богу похоже на iOS с её вызовами `sceneDidDisconnect()`, `sceneDidBecomeActive()`, `sceneWillResignActive()`, в определённые моменты жизненного цикла приложения, хотя трудно в event-driven архитектуре придумать что-нибудь новое):

```
public class BookkeeperActivity extends AppCompatActivity {
    private BookkeeperViewModel mViewModel;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_bookkeeper);
```

```
        mViewModel = new ViewModelProvider(this).get(
            BookkeeperViewModel.class);

        FragmentManager fm = getSupportFragmentManager();
        Fragment fragment = fm.findFragmentById(R.id.
            fragment_container);

        if (fragment == null) {
            fragment = new BookkeeperFragment();
            fm.beginTransaction()
            .add(R.id.fragment_container, fragment)
            .commit();
        }
    }
}
```

Метод `onCreate()` сохраняет экземпляр `BookkeeperViewModel`, который и будет в дальнейшем являться контроллером, загружает главный фрагмент `fragment_container` и переключат управление на него. Всё, с этого момента за работу приложения полностью отвечает фрагмент `BookkeeperFragment`. Данный класс получает от OS Android сообщение `onViewCreated` когда представление для него полностью готовою

```
    @Override
    public void onViewCreated(@NonNull View view, @Nullable Bundle
        savedInstanceState) {
        super.onViewCreated(view, savedInstanceState);

        findViews();
        setTrackingLabel();
        setListeners();
        setMonthYearButtonSince();
```

```
        setMonthYearButtonTill();
        showAmountsInfo();
    }
```

Далее он инициализирует все необходимые для своего экрана виджеты и в методе `showAmountsInfo()` пытается вывести на экран сколько финансов согласно базе данных было получено и сколько было потрачено начиная с определённой и заканчивая другой определённой датой.

Дата начала и дата окончания просмотра устанавливаются методами `setMonthYearButtonSince()` и `setMonthYearButtonTill()` соответственно. Метод `setTrackingLabel()` выводит на экран с какой даты по какую в данный момент будет показана информация или строку "На этот временной промежуток данных нет"если в базе отсутствуют какие-либо финансовые операции на данный промежуток времени или база пока что вообще пустая.

Сначала ищется самая ранняя финансовая операция путём выполнения SQL запроса:

```
"SELECT MIN(flow_date) FROM money_flows"
```

Так как эта операция асинхронная, то она выполняется не в основном потоке, и через переменную типа LiveData<Long> дожидается завершения запроса, не мешая основному потоку общаться с пользователем:

```
private void setTrackingLabel() {
    LiveData<Long> minMoneyFlowDate = mViewModel.getMinMoneyFlowDate
        ();

    minMoneyFlowDate.observe(getViewLifecycleOwner(), new Observer<
        Long>() {
        @SuppressLint("StringFormatMatches")
        @Override
        public void onChanged(Long minMoneyFlowDate) {
            if (minMoneyFlowDate == null || minMoneyFlowDate.
```

```
                longValue() == 0) {
                mTvTrackingStartLabel.setVisibility(View.GONE);
                mTvTrackingStartDate.setVisibility(View.GONE);
            } else {
                mTvTrackingStartLabel.setVisibility(View.VISIBLE);
                mTvTrackingStartDate.setVisibility(View.VISIBLE);

                Calendar calendar = Calendar.getInstance();
                calendar.setTimeInMillis(minMoneyFlowDate.longValue
                    ());

                mTvTrackingStartDate.setText(String
                .format(getString(R.string.tracking_start_date),
                getResources()
                .getStringArray(
                R.array.month_names_gentive)[calendar
                .get(Calendar.MONTH)],
                calendar.get(Calendar.YEAR)));

                calendar.setTimeInMillis(minMoneyFlowDate.longValue
                    ());
                mViewModel.setStartOfTrackingDate(calendar);
            }
        }
    });
}
```

После завершения запроса срабатывает наблюдатель-observer, который в зависимости от ответа или отображает на экране с какой даты имеются данные по движению финансов, или выводит строку "На этот временной промежуток данных нет".

Тут опять используется LiveData — если база данных не пустая, то посредством SQL запроса:

```
"SELECT␣*␣FROM␣money_flows␣WHERE␣flow_date␣BETWEEN␣:rangeLo␣AND␣␣:
    rangeHi␣ORDER_BY␣flow_date"
```

определяется какие записи по движению финансов попадают в промежуток от дата начала до дата окончания просмотра и

Так же на этом экране выводится кнопка "Добавить денежную операцию"и *только в случае если база данных не пустая и в заданном промежутке финансовые операции присутствуют* так же выводится кнопка "Детальная информация", нажав на которую можно посмотреть список всех денежных операций, которые были начиная с даты старта отслеживания и заканчивая датой окончания отслеживания, которые, как я уже говорил выше, устанавливаются в методах `setMonthYearButtonSince()` и `setMonthYearButtonTill()`.

За вывод детальной информации отвечает уже другой фрагмент с полностью своим экраном, который показывает лишь RecyclerView (в iOS ближайший аналог это `UICollectionViewController` и другие подобные из этого семейства) со вертикально прокручивающимся списком всех финансовых операций, которые удовлетворяют заданному ранее критерию.

При нажатии на элемент из этого списка запускается другой фрагмент, который предлагает или отредактировать этот элемент (изменить место проведения операции, или её дату, или её время), или изменить эту операцию с операции пополнения на операцию расхода, или наоборот, поменять статус с операции расхода на опреацию пополнения, или же вообще удалить этот элемент из базы.

Этот же фрагмент запускается с главного экрана при нажатии кнопки "Добавить денежную операцию", единственное различие между вызовами данного фрагмента с главного экрана или при нажатии на один из элементов списка финансовых операций за период — это наличие возможности удалить

операцию:

```
public BookkeeperMoneyFlowEditFragment(boolean isNewMoneyFlow,
    MoneyFlow moneyFlow) {
     mIsNewMoneyFlow = isNewMoneyFlow;
     mMoneyFlow = isNewMoneyFlow ? new MoneyFlow() : moneyFlow;
}
```

С главного экрана данный фрагмент вызывается с `true` в параметре `isNewMoneyFlow`, а с экрана, где выбирается уже существующий элемент из списка, с параметром `false`, поэтому для вновь создаваемой записи о финансовой операции доступна лишь кнопка "Сохранить", а для уже существующей в базе фрагмент `BookkeeperMoneyFlowEditFragment` показывает ещё и кнопку "Удалить"— нельзя ведь удалить *новую*) запись которая в данный момент только создаётся и в базе её пока ещё нет.

Удаление происходит по уникальному **id** записи:

```
mBtnMoneyFlowDelete.setOnClickListener(new View.OnClickListener() {
     @Override
     public void onClick(View v) {
          BookkeeperRepository.getRepository().deleteMoneyFlow(
              mMoneyFlow.getId());
          getActivity().getSupportFragmentManager().popBackStack();
     }
});
}
```

путём выполнения SQL запроса

```
"DELETE FROM money_flows WHERE id = :id"
```

в отдельном, не в главном потоке. Так как `id` у каждой записи уникальный (см. тут3.2), то удаляется именно та запись, которую пользователь и попросил удалить.

Что на главном экране, что на экране редактирования элемента из списка

показанных финансовых операций для ввода даты (а на экране редактирования элемента ещё и для ввода времени) вызываются встроенные готовые функции для ввода даты и времени из присутствующих в OS Android классов `DatePicker` и `TimePicker` соответственно.

После ввода даты обязательно производится проверка: дата начала или окончания просмотра не может быть раньше даты, с которой только начинается фиксирование финансовых операций в базе (я выше уже говорил что дата начала этих фиксаций определяется в методе `setTrackingLabel()` основного, главного фрагмента), а так же дата начала просмотра не может быть позже даты окончания просмотра, все эти проверки обязательно производятся и при некорректной даты не допускается.

Так же проверяется чтобы все поля были заполнены: и место проведения операции, и дата, и время, и сумма. Операции проведённые неизвестно где, неизвестно когда, неизвестно во сколько и на неизвестно какую сумму в базу не попадут:

```
mBtnMoneyFlowSave.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        if (mTvMoneyFlowPlace.getText().toString().isEmpty()) {
            Toast.makeText(
            getActivity(), R.string.no_place_provided, Toast.
                LENGTH_LONG).show();
        } else if (mTvMoneyFlowDate.getText().toString().isEmpty())
            {
            Toast.makeText(
            getActivity(), R.string.no_date_provided, Toast.
                LENGTH_LONG).show();
        } else if (mTvMoneyFlowTime.getText().toString().isEmpty())
            {
            Toast.makeText(
```

```
            getActivity(), R.string.no_time_provided, Toast.
                LENGTH_LONG).show();
        }
        if (mTvMoneyFlowAmount.getText().toString().isEmpty()) {
            Toast.makeText(
            getActivity(), R.string.no_account_provided, Toast.
                LENGTH_LONG).show();
        } else {
            mMoneyFlow.setName(mTvMoneyFlowPlace.getText().toString
                ());

            if (mCheckMoneyFlowIncoming.isChecked()) {
                if (mMoneyFlow.getAmount() < 0) {
                    mMoneyFlow.setAmount(-mMoneyFlow.getAmount());
                }
            } else {
                if (mMoneyFlow.getAmount() > 0) {
                    mMoneyFlow.setAmount(-mMoneyFlow.getAmount());
                }
            }
            if (mTvMoneyFlowAmount.getText().toString().indexOf('.')
                == -1) {
                mMoneyFlow.setAmount(mMoneyFlow.getAmount() * 100);
            }

            if (mIsNewMoneyFlow) {
                BookkeeperRepository.getRepository().insertMoneyFlow
                    (mMoneyFlow);
            } else {
                BookkeeperRepository.getRepository().updateMoneyFlow
                    (mMoneyFlow);
            }
            getActivity().getSupportFragmentManager().popBackStack()
```

```
                    ;
            }
        }
});
```

Глава 4

Заключение

Итогом данной моей дипломной работы явилась разработка приложения **"Bookkeeper"**для удобного учёта моих личных финансов

В процессе разработки я понял что в контексте данного приложения необходимо сделать прямо сейчас, что желательно сделать после и что **обязательно** необходимо сделать после для дальнейшего улучшения функциональности и удобства пользования данным приложением.

На данный момент приложение считаю оконченным для представления в качестве дипломной работы, но обязательно буду его улучшать, дополнять и рефакторить.

Я не писатель и с огромным трудом смог набить тут тридцать три страницы, на 50+ я уже не вытянул никак, хотя и старался. Я мог бы "добить"оставшисе до 50 страницы запостив сюда кучу скриншотов и прочей "воды", но я считаю что это было бы просто нечестно. Поэтому рассказал всё, что мог рассказать, ни одного скриншота не прикладывал. А больше ничего рассказать, увы, не могу.

И под iOS я диплом сделать не смог по причине, про которую я уже говорил выше, а именно в разделе про `Технологический стек` (2.1)

Литература

[1] https://developer.apple.com/documentation/coredata/ — Apple Core Data

[2] https://www.sqlite.org/index.html — SQLite site

[3] https://developer.android.com/jetpack/androidx/releases/room — Android Room

[4] https://habr.com/ru/articles/713518/ — про Android Room на Хабре

[5] Филлипс Б., Стюарт К., Марсикано К.*Android. Программирование для профессионалов. 3-е изд.* СПб.: Питер,2017. – 688 с.: ил. – (Серия «Для профессионалов»).