

ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - ВАРНА
ФАКУЛТЕТ ПО ИЗЧИСЛИТЕЛНА ТЕХНИКА И АВТОМАТИЗАЦИЯ



Катедра „Софтуерни и интернет технологии“

КУРСОВ ПРОЕКТ

Тема:

Проектиране и реализиране на система за отчитане
на ежедневни разходи

Изготвили:

Михаел Бердж Агопян ФН: 18621765

Георги Грозданов Георгиев ФН: 18621746

Специалност: Софтуерни и интернет технологии

ТУ Варна, 10.01.2022 г.

Ръководител:

/х. ас. Д.Димитров/

Съдържание

1. Увод	4
2. Фаза на проучване	5
2.1 Конкурентен анализ	5
2.2 Проучване на потребители – Google Form.....	10
3. Анализ на проблема	13
4. Избор на технологии	15
4.1 Мобилна платформа (Android OS).....	15
4.1.1 История.....	15
4.1.2 Защо мобилна платформа на Андроид?	15
4.2 Интегрирана среда за разработка (Android Studio IDE).....	16
4.2.1 История.....	16
4.2.2 Защо избрахме Android Studio IDE?	16
4.3 Език за програмиране (JAVA 8).....	17
4.3.1 История.....	17
4.3.2 Защо избрахме JAVA 8?	17
4.4 База данни (SQLite)	18
4.4.1 История.....	18
4.4.2 Защо избрахме SQLite?	18
4.5 Room Persistence Library	19
4.6 Система за контрол на версиите (GitHub).....	19
4.7 Концептуален модел	20
5. Фаза на планиране.....	21
5.1 Персони (Personas)	21
5.2 Storyboard	23
5.3 UML Diagram	24
5.4 Essential Use Case Diagram.....	25
5.5 Карта на съпричастност	26
5.6 Mindmap	27
5.7 Sitemap.....	28
5.8 Прототип	29
5.9 Coremodels.....	33
6. Фаза на проектиране	35
6.1 Moodboard	35

6.2 Лого.....	36
6.3 Модел на базата данни.....	37
7. Заключение	39
8. Приложение	40
8.1 Entity	40
8.2 DAO (Data Access Object)	43
8.3 Repository	45
8.4 Viewmodel	51
8.5 LiveData	54

1. Увод



В днешно време, управлението на финансите е доста важна част от нашето битово ежедневие. С всяка година ежедневието ни става все по-забързано, появяват се нови и различни отговорности и сякаш се налага да вършим всичко с по-голяма скорост и прецизност от преди. За щастие, благодарение на бързото развитие на технологията, ни се предоставят все повече нови възможности и функционалности, както имаме и свободен достъп до все повече информация, което създава предпоставки за по-високо качество, продуктивност и увереност в нашите действия.

Навременната, актуална и точна информация е от особено важно значение за управлението на нашите финанси. До голяма степен финансовото ни състояние се определя и от стратегическия финансов план, който сме изградили, а не само от получаването на приходи. От изключително голяма важност е човек, да умее да управлява правилно и най-вече ефективно своите разходи. Важно е да бъдат избегнати ненужните, непродуктивни действия и също така е от изключителна важност намаляването на шансовете за допускане на грешка до минимум, тъй като това би означавало загуба на финансови средства за нас. Скоростта на калкулацията на сметките трябва да бъде по възможност най-бърза и правилна. Актуалната и навременна информация също е от ключово значение, тъй като винаги бихме искали да разполагаме с наличното към настоящия момент, а не да се изчисляват приблизителни суми.

Ефективността при управлението на личните ни финанси, би могла да бъде постигната, чрез използването на определени инструменти за това. Като един от най-удобните и ефективни начини за постигане на тази съответна цел, е чрез използването на мобилно приложение, за управление на разходите и приходите. Такъв тип мобилно приложение, ще ни предоставя необходимата сигурност, точност, коректност и актуална, и навременна информация във всеки един момент. С този инструмент си гарантираме повишаването на продуктивността и ефективността при изчисляването на нашите разходи и приходи.

2. Фаза на проучване

2.1 Конкурентен анализ

Критерии	<p>Monefy – Budget Manager and Expense Tracking App на <i>Reflectly</i></p> 	<p>Money Manager Expense & Budget на <i>Realbyte Inc.</i></p> 
Информация за състоянието на системата	<p>Системата предоставя семпъл и ефективен интерфейс с добре групирани елементи, който позволява интуитивна работа с нея дори за потребители, които използват подобно приложение за пръв път. Основна функционалност е добавянето на записи за разходи и приходи и категоризирането им. Предоставя лесен начин за справки за период от време. Функционалности като синхронизация между множество устройства, задаване на валути, както и козметични промени като Dark mode не са включени в безплатната версия. Има функция за експорт на записите в избран формат</p>	<p>Системата предоставя възможност за отчитане на приходи и разходи по категории, както и за прехвърляне на средства от една сметка в друга. Записите могат да се представят графично за избран период от време. Всички функционалности са включени в безплатната версия, като платената единствено премахва рекламите. Освен начини за експорт на записите в избран формат, системата позволява изпращане по имейл, като автоматично прикачва .sqlite файл, който може да бъде restore-нат на друго устройство. Друг начин за backup е синхронизиране с Google Drive с възможност за restore от същия</p>
Съвпадение на нуждите на реалния свят с предоставянит е от	<p>Системата предоставя основни функционалности като избиране на вида на разходите и приходите, но липсват подвидове, напр.</p>	<p>Приложението предоставя повечето функционалности, с които да покрие потребителските нужди, но липсват подкатегории на</p>

приложението	има разходи за автомобил, но няма начин потребителят да конкретизира точно за какво е разхода – За ремонт? Гориво? И т.н. Има начин за добавяне на нови категории, но в платената версия, и дори това не решава проблема с липсата на йерархия (подкатегории). Липсата на избор на валути може да отблъсне потребители, които имат сметки в различни валути.	категориите приходи и разходи. Функционалността за добавяне на нови категории е полезна, но не заменя нуждата от йерархична структура.
Потребителски контрол и свобода на действията	Потребителите имат свободата да избират категория на разходите/приходите, да избират интервал от време за справки, но въпреки че системата предоставя меню с около 20 опции, повечето от тях (за синхронизация, backup, избиране на валута, опции за по-голяма сигурност) остават заключени в безплатната версия	Потребителите имат свободата да избират категория на разходите/приходите, да избират интервал от време за справки, да избират цвetoва гама на интерфейса на приложението. Има налични всички валути и за всяка сметка те могат да се променят.
Консистентно представяне на информацията и стандарти	Информацията е нагледно представена в основния, начален екран на приложението. Разходите и приходите са представени графично в кръгова диаграма с празен център с текст в него, която не е идеалният вариант за целта и би било добре да има поне опция за представяне с друг вид диаграма	Името на таба в navigation bar за преглед на списъка и добавяне на нов приход/разход е Transactions, но тъй като е дълга дума е съкратено на „Trans.”, което не е удачно за толкова ключов елемент от интерфейса. Записите на същия основен екран са представени в списък, но както в него, така и на други места в приложението текстът на

		<p>места е труден за четене, в сив вместо черен цвят, с малък размер и шрифт, който в тази ситуация прави текста още по-труден за четене</p> <p>Графично представените справки са под формата на кръгова диаграма с добре обозначено това кой цвят какво преставлява</p>
Предотвратява не на грешки и тяхното прихващане	Системата не позволява добавяне на записи без минимална въведена информация (в случая сумата на прихода/разхода – категорията му се избира предварително)	Системата не позволява добавяне на записи без минимална въведена информация (в случая сумата на прихода/разхода, категорията и разхода)
Осигуряване на интерфейс пред действие по памет	Основните функционалности са лесно достъпни, нагледни и говорят за себе си	Основните функционалности са лесно достъпни, нагледни и говорят за себе си
Гъвкавост и ефективност	Липсват подкатегории за приходите и разходите, и няма функционалност да се добавят такива. Дори в платената верия, която предоставя множество други подобрения, на фона на липсата на подкатегории те изглеждат тривиални. Има пълна гъвкавост при отчетите за период – за деня/седмицата/месеца/година, както и за зададен интервал от-до дата. Липсата на валути в безплатната версия и това намалява гъвкавостта ѝ за една част от потребителите	Липсват подкатегории за приходите и разходите, и няма функционалност да се добавят такива. Налични са всички валути. Има пълна гъвкавост при отчетите за период – за деня/седмицата/месеца/година, както и за зададен интервал от-до дата. Броят сметки, които могат да се добавят е ограничен в безплатната версия. Няколкото начина за баскир дават свобода на избор и са обяснени нагледно в секцията за помощ. Към запис за приход/разход може да се

		добави снимка
Минималистичен дизайн	<p>Приложението има минималистичен, на места твърде семпъл дизайн. Анимации при интеракция с основните бутони за добавяне на приход или разход липсват, те са представени като големи кръгове с оцветен border и бяла сърцевина с текст, което не е оптимално имайки в предвид предназначението им. На места са използвани излишни анимации като леко подскачащ бутон при отваряне на форма. Някои от иконките, с които са представени видовете разходи са неудачни, като например живачен термометър за разходи за здраве</p>	<p>Приложението има минималистичен дизайн, но цветовете, различни от бял фон с черен текст или иконки със сиви очертания почти не се срещат. На основния екран текста за стойността на приходите е в тъмносин цвят вместо по-интуитивния зелен. Разходите са в червено, но същото червено се използва като основен цвят на приложението за обозначаване на текущия таб и на други места и основния бутон за добавяне на запис също е в червено, при положение че записа може да бъде не само разход, но и приход или трансфер. Липсва логика в избора на цветовете. На почти половината от екраните има реклами, за чиито премахване може да се заплати. Рекламите не са удачни за такъв вид приложение. Текстът на места е труден за четене. Анимации липсват, има бързи transition-и при превключване между табове, но като цяло усещането за интерактивност липсва</p>

Обратна връзка	Възможно е единствено оценяване на приложението в Google Play Store, за което има бутон в едно от менютата на приложението	Освен оценяване на приложението в Google Play Store, то има и бутон Feedback, който води до екран с уебсайта и имейла на компанията-разработчик и опции за Feedback, при което се избира приложение, което да се отвори, за да продължим. При отваряне с Gmail автоматично се зарежда темплейтно писмо с уникален код и информация за версията и устройството, но в дългия списък с възможни приложения влизат и такива като Viber, които при избиране се отварят и не водят до нищо.
Помощ и документация	При първо добавяне на запис има напътствия под формата на подсказки под полета и бутони, но след това липсва начин да се видят помощни ресурси	Системата няма въвеждащи подсказки при първо използване, но има секция Help, в която има описани промените за всеки ъпдейт, както и параграфи с често срещани въпроси, които нагледно с помощта и на скрийншоти обясняват как да ползваме приложението и разясняват за какво служи всеки елемент от интерфейса

2.2 Проучване на потребители – Google Form

Изследването на потребителските нагласи за потребителското изживяване е ключова част от разработката на софтурния ни продукт и използва метода за UX research *Customer feedback* (отзиви на клиентите/потребителите). Бе извършено с помощта на Google Forms за събиране на отговорите на потребителите и извеждане на резултатите в графичен вид.

Изследваната група се състои от 15 студенти от 4 курс от специалност Софтуерни и интернет технологии в Технически университет – Варна, които отговориха на 5 въпроса със затворен отговор. Първите 4 от тях бяха задължителни, а 5-ят получи само някои от анкетираните.

Изследването даде следните резултати:

1. На въпроса колко често използват системи за следене на приходите и разходите си (Фиг. 1.1), малко над $\frac{1}{4}$ от анкетираните заявиха, че ги използват ежедневно. Същият процент никога не са използвали подобни системи.



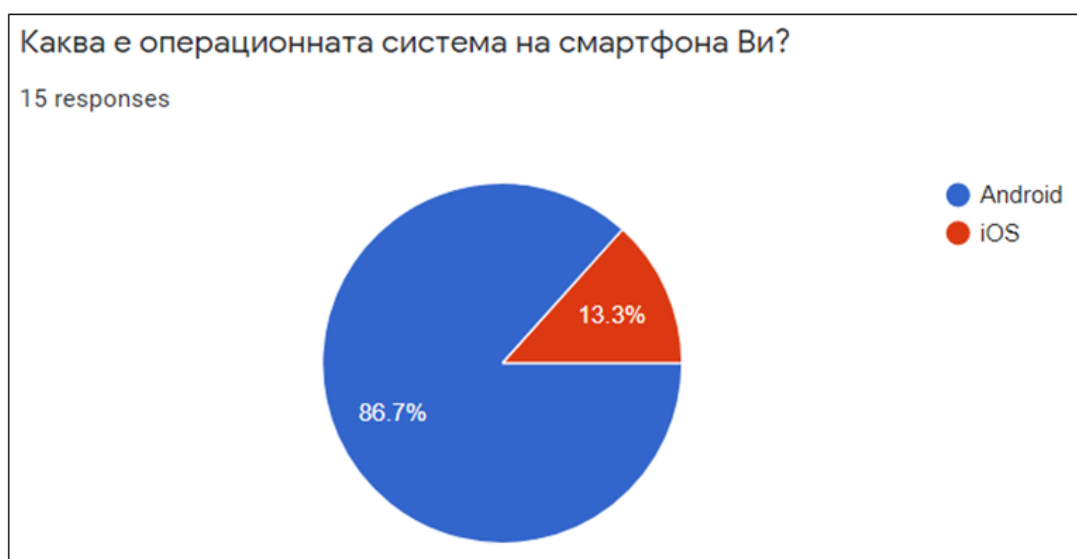
Фиг. 1.1 Резултати от проучване

2. На въпроса коя платформа е най-подходяща за едно приложение за отчитане на приходи и разходи (Фиг. 1.2), 4/5 от анкетираните посочват мобилната.



Фиг. 1.2 Резултати от проучване

3. На въпроса каква операционна система използва телефонът им (Фиг. 1.3), над 85% посочват Android.



Фиг. 1.3 Резултати от проучване

4. На въпроса дали са склонни да заплатят за подобно приложение (Фиг.1.4), анкетираниите имаха възможност да посочат и повече от един отговор. Преобладаващият отговор е „Не бих платил(а)“ с 40%. Малко над $\frac{1}{4}$ биха заплатили, ако приложението е досатъчно

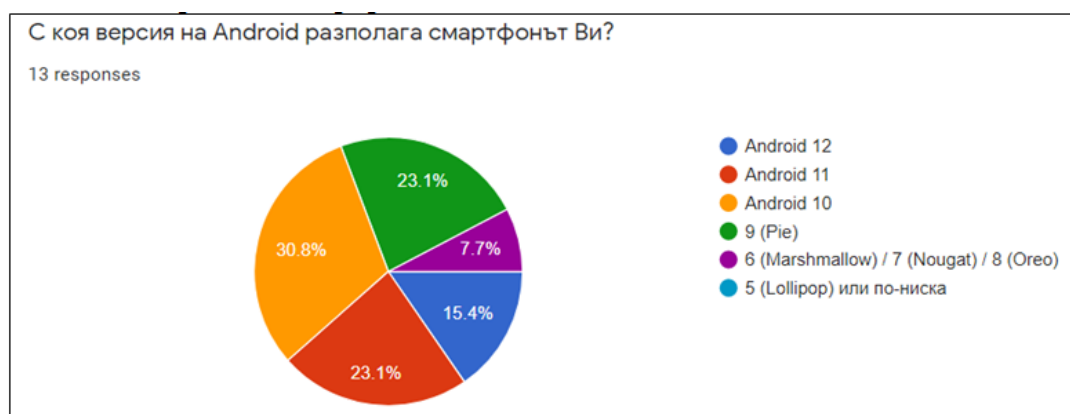
качествено или ако чрез заплащане се премахват рекламите. 1/5 биха заплатили, ако това би отключило нови функционалности.



Фиг. 1.4 Резултати от проучване

Анкетираните, отговорили, че операционната система на смартфона им е Android, получиха допълнителен въпрос.

- На въпроса коя е версията на Android, която използват (Фиг. 1.5), с най-голям дял (30%) се откроява Android 10. На второ място се нареждат Android 9 Pie и Android 11 с по 23%. Никой от анкетираните не използва версия 5 Lollipop или по-ниска.



Фиг. 1.5 Резултати от проучване

3. Анализ на проблема

С проектирането и реализирането на мобилно приложение за управление на личните финанси, бихме искали до някаква степен да автоматизираме и улесним процеса по калкулация и изчисляване на разходи и приходи, както и да се погрижим за безопасното и сигурно съхранение на тази информация. Важно е информацията да бъде винаги лесно достъпна, да бъде актуална, точна, сигурна и правилна. Представянето на статистика под формата на кръгови диаграми също е от голямо значение, с цел по-лесното възприемане на информацията от човешкото око. Приложението е съсредоточено към повишаване на продуктивността и ефективността при процеса на управление на финансите, с цел помагане и улесняване на усилията на крайните потребители.

Първия проблем, който се решава, използвайки това мобилно приложение е свързан с правилните изчисления на разходи или приходи, като това е от особено важно значение, тъй като по този начин се елиминира шанса за грешни суми и съответно крайните потребители се предпазват от некоректни или неточни бъдещи инвестиции или планове.

Следващата полза от използването на тази система за отчитане на ежедневни разходи или приходи е именно дигиталния процес. Спестяват се вложените на ръчни усилия, за пресмятането на сумите и съответно записването и съхранението им на хартиен носител, което е несигурно, непродуктивно и неефективно. Чрез използването на технологията, информацията се съхранява на сигурно място, като винаги е налична във всеки един момент. Автоматизира се процеса на пресмятане, като това спестява доста усилия и време на крайните потребители, като най-важната особеност, е че се повишава продуктивността. Друга положителна черта, е че се изключва използването на хартиени носители, следствие на това се решават проблеми свързани със съхранението на тези носители, конфиденциалността на информацията и съответно ненавременното наличие на информацията при нужда.

Друга отличителна особеност на мобилното приложение е, че е лесно за използване с опростен интерфейс, включващ всички необходими функционалности за използване и управление на нашите финанси. Системата предоставя възможности за категоризация на нашите приходи и разходи, и съответно подкатегоризация с наличието на съответни икони, индикиращи

целта или по-скоро визуалното представяне на сферата или категорията, към която попада нашия приход или разход.

Предоставят се възможности за създаване на различни сметки или пера, в различни валути за проследяване на определени финансови средства в различни категории или сфери. Като има възможност за представяне на информацията за разходите и приходите като статистика за определени сметки под формата на кръгови диаграми. Допълнителна функционалност е наличието на търсене и филтрация на информацията, което значително ускорява процеса по търсене на конкретните сметки или осъществени транзакции (приходи или разходи).

4. Избор на технологии

Реализацията на нашата система за управление на личните финансови средства, сме избрали следните технологии:

4.1 Мобилна платформа (Android OS)

4.1.1 История

Android е операционна система на Google Inc. за мобилни устройства, базирана на ядрото на Linux. Тя е създадена, поддържана и развивана първоначално от Android Inc., която е купена от Google Inc. през 2005 г. Пускането ѝ на пазара на 5 ноември 2007 г. е съпроводено с основаването на Open Handset Alliance – консорциум от хардуерни, софтуерни и телекомуникационни компании за развиването на отворени стандарти при мобилните устройства. Google пуска фронтенд кода под свободен лиценз. За развитието на Android се грижат голям брой софтуерни разработчици, които създават така наречените „apps“ (Applications) – малки приложения, които разширяват функционалността на системата. Приложенията могат да бъдат сваляни от различни сайтове в интернет или от големи онлайн магазини, като Android Market (впоследствие преименуван на Google Play) – магазинът на Google. По данни към юни 2018 г. за Android има над 3 300 000 приложения. Приложенията се пишат предимно на Java, Python или Ruby.



4.1.2 Защо мобилна платформа на Андроид?

Реализацията на нашата система, решихме да я изградим под формата на мобилно приложение за Андроид Операционна система. Избрахме мобилната платформа тъй като в днешно време рязко нараства броя на потребители, които използват смартфони и съответно все по-голяма част от потребители предпочитат използването на мобилни приложения вместо използването на настолен компютър или лаптоп, тъй като това е доста по-удобно и ефективно в някои случаи. Смартфоните стават все по-популярни като все повече потребители инвестират време използвайки тях, вместо другите платформи. Избрахме да реализираме приложението на Android OS,

тъй като в днешно време за крайните потребители е много по-лесно откъм финансова гледна точка да се сдобият с Андроид телефон отколкото iPhone с неговия iOS. Към Юли 2021 г. повече от 70% от мобилните устройства са поддържани на Android OS. Следователно пазарът тук е доста по-голям. Друга негативна черта за реализацията на iOS е бариерата за навлизане за разработка на приложение. Наложително е използването на скъпа техника, която поддържа MacOS както и наличие на iPhone за истинска тестова среда.

4.2 Интегрирана среда за разработка (Android Studio IDE)

4.2.1 История

Android Studio е официалната интегрирана среда за разработка (IDE) за операционната система Android на Google, изградена върху софтуера IntelliJ IDEA на JetBrains и проектирана специално за разработка на Android. Предлага се за изтегляне на базирани на Windows, macOS и Linux операционни системи или като абонаментна услуга през 2020 г. Той е заместител на Eclipse Android Development Tools (E-ADT) като основна IDE за разработка на приложения за Android.



4.2.2 Защо избрахме Android Studio IDE?

Избрахме разработката на нашето мобилно приложение да се осъществи на **Android Studio IDE**, защото е софтуер разработен и препоръчан за използване от **Google**, както и факта че предоставя функционалности като Code Completion, поддържа Gradle, възможност за преглеждане едновременно на XML кода и самия XML Layout на приложението, докато при Eclipse това не е възможно, трябва да се избере само единия вариант при проектиране на интерфейс и да се сменя постоянно между двете възможности, за да се проследяват промените по интерфейса и XML кода.

4.3 Език за програмиране (JAVA 8)

4.3.1 История

Java или Джава е обектно ориентиран език за програмиране, разработен от Sun Microsystems и пуснат в употреба през 1995 година, като част от Java платформата. Впоследствие се появяват множество други реализации включително от GNU, Microsoft, IBM, Oracle и други технологични доставчици. Изходният код, написан на Java, не се компилира до машинен код за определен микропроцесор, а се компилира до междинен език - така нареченият байткод. Байт кодът не се предава за директно изпълнение от процесора, а се изпълнява от негов аналог – виртуален процесор, наречен Java Virtual Machine (JVM). Java е език за програмиране от високо ниво с общо предназначение. Синтаксисът му е подобен на C и C++, но не поддържа много от неговите възможности с цел опростяване на езика, улесняване на програмирането и повишаване на сигурността. Програмите, написани на Java, представляват един или няколко файла с разширение .java. Тези файлове се компилират от компилатора на Java – javac до изпълним код и се записват във файлове със същото име, но различно разширение .class. Клас-файловете съдържат Java байткод инструкции, изпълним от виртуалната машина.



4.3.2 Защо избрахме JAVA 8?

Избрахме за разработка на нашето мобилно приложение да използваме **JAVA 8**, тъй като към настоящия момент разполагаме с повече знания и умения на този програмен език и бихме искали да се съсредоточим повече към разработката на приложението вместо изучаването другия популярен програмен език **KOTLIN**. Той е сравнително нов език, който тепърва навлиза и поддържа ООП и Функционално програмиране, но времето за компилация или Compilation Time на JAVA е около 15-20 % по-бързо, също така има много повече информация и ресурси на разположение при проблеми възникнали на JAVA, тъй като това е програмен език на 26 години. Избрахме версия 8 на JAVA, защото разполагаме с голям брой библиотеки, които можем да използваме наготово, тъй като от версия 9 и нагоре, навлиза

концепцията за модулно програмиране и голяма част от библиотеките липсват и се налага ръчното изтегляне и конфигуриране. Ограничаваме ме се откъм функционалности като например, „Ламбда изрази” и „променливи с автоматичен тип”, но не е нещо изключително фатално.

4.4 База данни (SQLite)

4.4.1 История

SQLite е релационна база данни с отворен код поддържаща стандарта SQL. Реализирана е като библиотека към приложенията, а не като самостоятелно работеща програма. Използва се в продукти, разработвани от Adobe, Apple, Mozilla, Google и др. Концепцията на SQLite е цялата база да бъде съсредоточена в един-единствен файл. Това я прави база данни без сървърен процес, особено подходяща за използване в мобилни устройства, таблети и софтуер, където е невъзможно поддържането на сървърен процес. SQLite е бърза и надеждна база данни, затова е избрана за база данни по подразбиране в OpenOffice Base, LibreOffice Base, както и в езиците за програмиране PHP, Ruby on Rails и други. CMS системата Drupal също работи с SQLite.



4.4.2 Защо избрахме SQLite?

За нашите цели, **SQLite база данни** идеално се вписва, поради факта, че ние създаваме за момента офлайн мобилно приложение с локална база данни, а не система с достъп до сървър. **SQLite** е free open-source и не е нужна инсталация на допълнителен софтуер за използването на тази технология. Също така, тази база данни е доста лесна за използване, бърза е и е значително малка и лека като размер, като е нужно само импортирането на определени библиотеки за използването на тази технология.

4.5 Room Persistence Library

Android Room Persistence Library е един от наборите от библиотеки, предоставени от Android Jetpack, за да помогне на разработчиците да следват най-добрите практики, като същевременно елиминират стандартните кодове и намаляват фрагментацията. В приложенията за Android често се налага да съхраняваме структурирани данни и запазването на тези данни локално винаги е добра идея. По този начин, ако нашето приложение се срина или се рестартира, данните няма да бъдат загубени и тези данни могат да бъдат извлечени по-късно. Особено когато трябва да покажем някаква подходяща информация на потребителя, която трябва да присъства дори при липса на мрежа, запазването на данни в приложението е много полезно.



Room Persistence Library

Part of Android Jetpack

4.6 Система за контрол на версиите (GitHub)

GitHub е уеб базирана услуга за разполагане на софтуерни проекти и техни съвместни разработки върху отдалечен интернет сървър в т.нар. хранилище (software repository). Базира се на Git системите за контрол и управление на версиите. Услугата може да бъде както платена за частни проекти, така и безплатна за т.нар. проекти с общодостъпен код, като и в двата случая потребителите могат да ползват всички възможности на услугата. Към май 2011 г. GitHub се счита за най-популярния сайт за разполагане на съвместни проекти с общодостъпен или наречен още отворен код. Компанията GitHub Inc. е основана през 2008 г. със седалище Сан

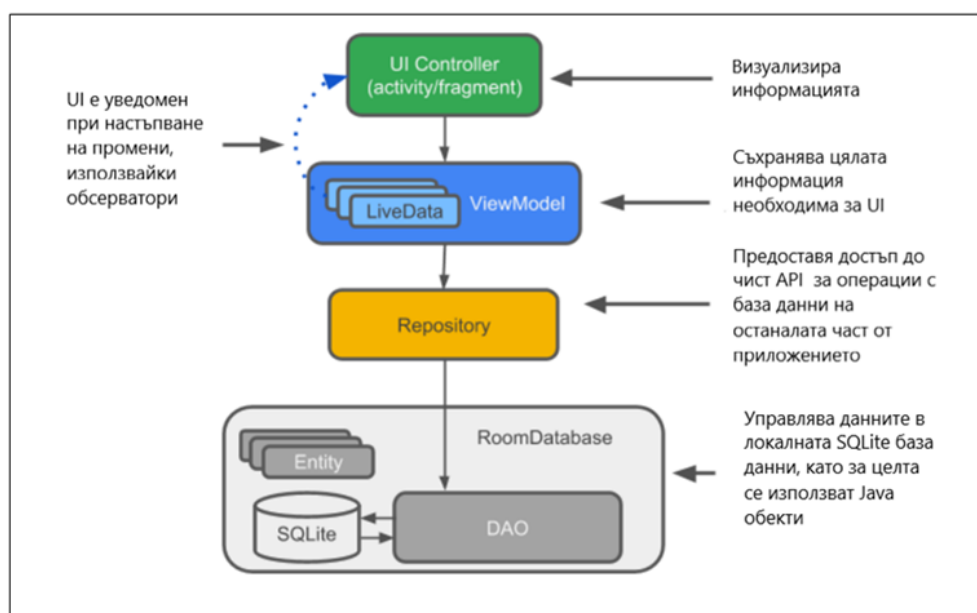


Франциско, Калифорния, САЩ. През юли 2012 г. компанията получава 100 милиона щатски долара първокласно финансиране, основно от компанията Andreessen Horowitz.

Позволява на много програмисти да работят едновременно върху един и същ проект, без да си пречат взаимно, като се запазва история на всички направени промени по текущия проект, като се разполага и със функционалности за разрешаване на конфликти при работа върху едни и същи файлове.

4.7 Концептуален модел

Операционната система Android управлява ресурсите агресивно, за да работи добре на огромен набор от устройства и понякога това е предизвикателство да се изградят стабилни приложения. Компонентите за архитектурата на Android предоставят насоки за архитектурата на приложението, с библиотеки за общи задачи като управление на жизнения цикъл и запазване на данните. Архитектурните компоненти ни помагат да структурираме приложението си по начин, който е стабилен, тестван и поддържан с по-малко ръчно написан код. Архитектурните компоненти предоставят прост, гъвкав и практичен подход, който ни освобождава от някои често срещани проблеми, за да можем да се съсредоточим върху изграждането на страхотни приложения.

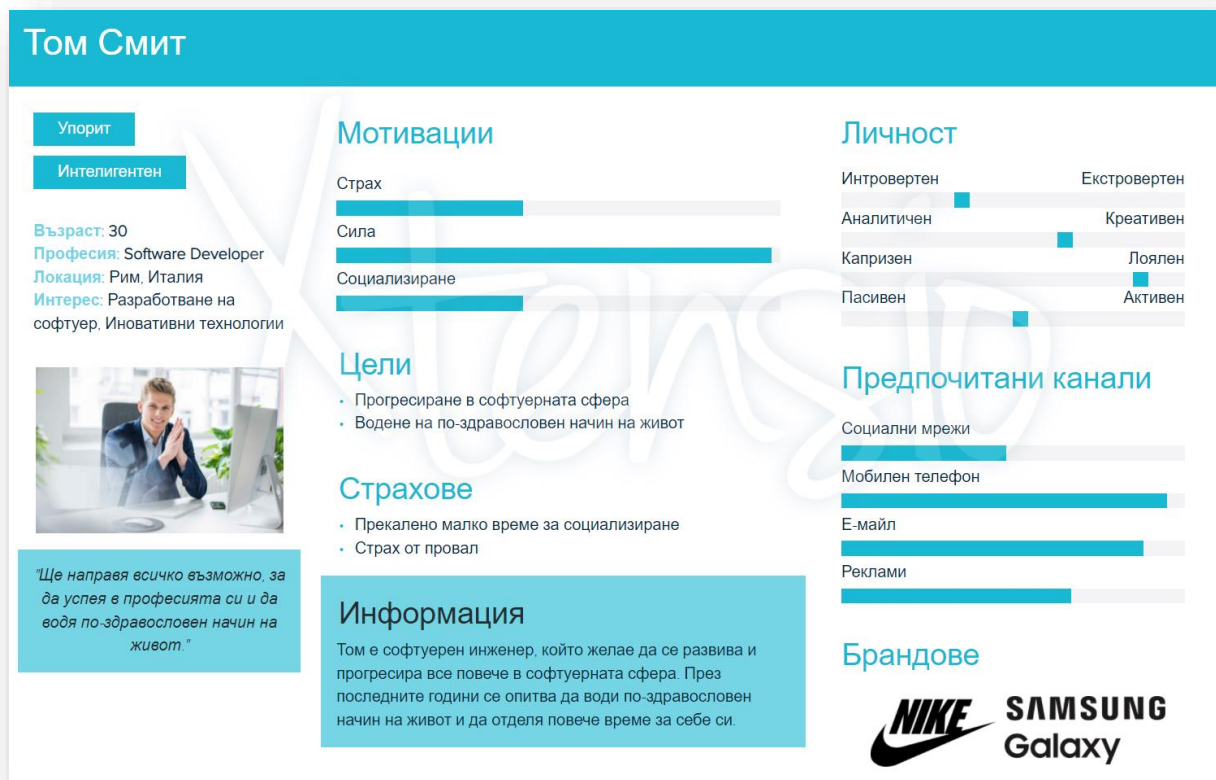


5. Фаза на планиране

5.1 Персони (Personas)

На етапа на проектирането на системата за отчитане на ежедневни разходи и приходи е изключително важно да бъдат идентифицирани персоните, за потребителят който най-лесно бихме могли да привлечем и съответно този който най-трудно бихме могли да привлечем, тъй като това би ни дало отговори на доста важни въпроси и съответно би определило и бъдещото развитие и разработване на нашия продукт.

Този тип инструмент ни позволява да опознаем аудиторията, за която разработваме продукта на едно доста по-добро ниво, като се стремим да се съобразяваме с техните интереси и нужди, и съответно да предоставим едно възхитително потребителско изживяване. Колкото по-детайлни и информативни са персоните, толкова по-лесно ще можем да анализираме и да удовлетворим нуждите на крайните потребители.



Ричард Джонсън

Общителен

Дисциплиниран

Възраст: 65

Професия: Пенсиор

Локация: Мадрид, Испания

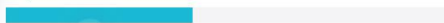
Интерес: Четене на литература,
Градинарство



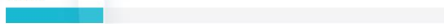
"Наслаждавайте се на живота"

Мотивации

Страх



Сила



Социализиране



Цели

- Желание да посещава други страни
- Да научава нови интересни факти

Страхове

- Недостатъчен опит с новите технологии (смарт телефон, компютър, интернет)

Информация

Ричард е от по-възрастното население, и в момента любимите му интереси са свързани с четене на литература и градинарство.

Личност

Интровертен

Екстравертен

Аналитичен

Креативен

Капризен

Лоялен

Пасивен

Активен

Предпочитани канали

Социални мрежи

Мобилен телефон

Е-майл

Реклами

Брандове



5.2 Storyboard

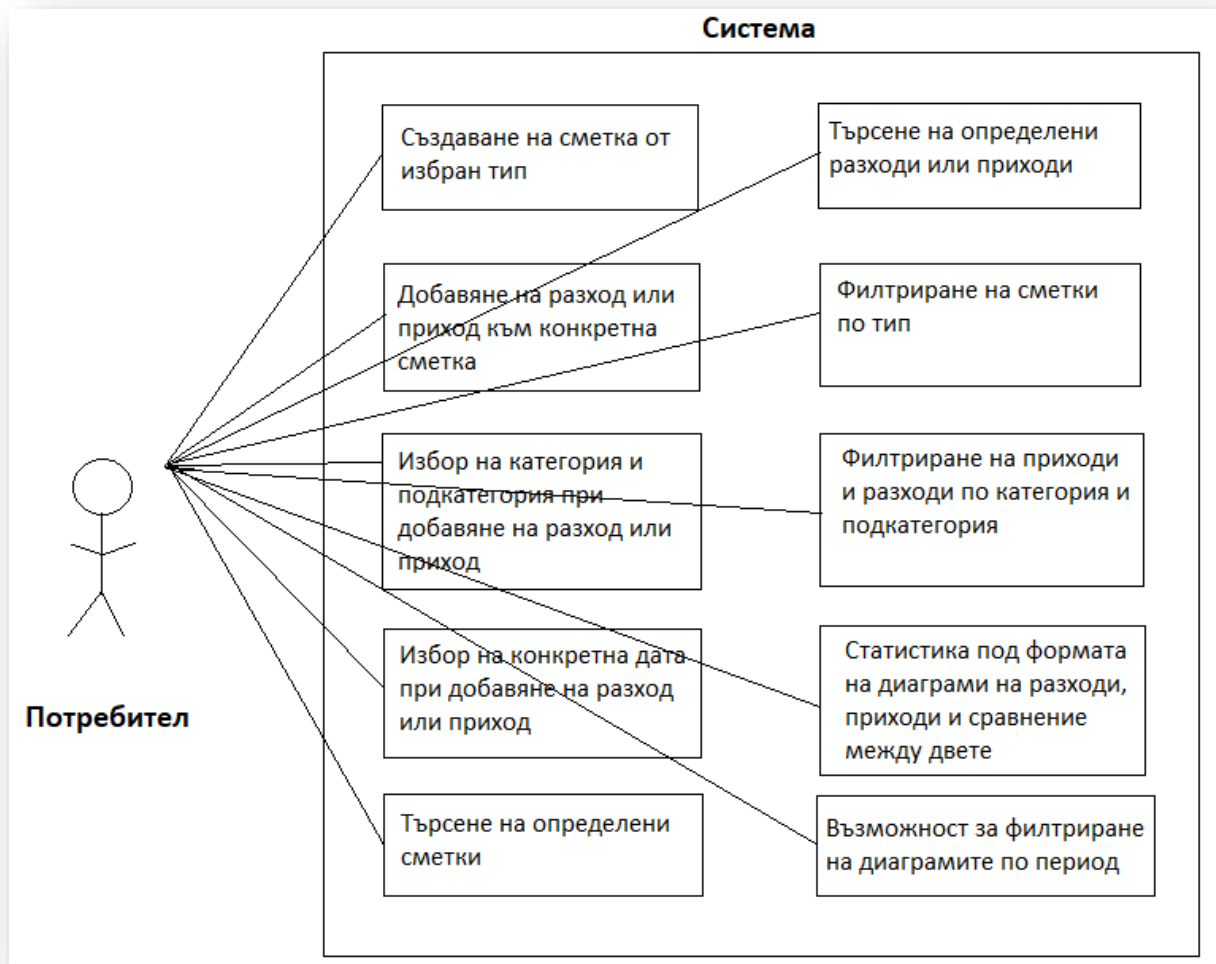
Създаването на история на потребителя е много важно, защото по този начин може да се дефинират ключовите моменти при създаването на дизайна. Storyboard-а представлява последователност от кадри, които служат за илюстриране на история. Изображенията се подреждат заедно, за формират същността на даден проблем, описан чрез скици. Общо казано това инструмент, с който може визуално да се прогнозира и изследва опита на потребител с даден продукт. Това би дало обратна връзка за това по какъв начин потребителя ще взаимодейства с продукта във времето, като се дадат ясни насоки.

Представеният по долу Storyboard ни илюстрира и показва, че потребителят е решил да вземе важно решение за управлението на своите лични финанси, и се е насочил към търсенето на мобилно приложение за тази цел. След като започва да използва функционалността на приложението **Pocket Finances**, потребителят бързо осъзнава, че това е подходящото средство за тази цел. С използването на мобилното приложение потребителят би могъл да има възможност да проследява своите разходи и приходи по всяко време и да разполага с актуална информация, независимо от мястото и времето на ситуацията.



5.3 UML Diagram

Този тип диаграма е също част от съвкупността от инструменти, които бихме могли да използваме при начална идентификация на изискванията и функционалностите, като разликата тук е, че имаме по-добро визуално представяне, с цел улесняване разбирането на системата от крайните потребители. Тук заявяваме, кой е нашият главен актьор, в нашия случай това е само един и това е крайният потребител, който ще използва системата, за удовлетворяване на своите нужди. Системата е другият актьор, която е представена на диаграмата като правоъгълник, който обхваща цялата функционалност и осигурява нейното възпроизвеждане, при инициране на действие от страна на потребителя.



5.4 Essential Use Case Diagram

При разработването на един софтуерен продукт, е от изключителна важност да бъдат уточнени детайлно изискванията и функционалностите от клиентите, още при началния етап на проектиране. За тази цел инструменти като Essential Use Case Diagram, са подходящи, за да се нахвърлят базовите изисквания и функционалности, както и кои са ключовите ни актьори и достъп до каква функционалност ще имат те. Този етап не бива да се пренебрегва тъй като липсата на пълна информация за изискванията на клиентите води до удължаване на крайните срокове, загуба на ресурси като пари и време, и съответно разочарование от страна на клиента.

Функционалност	Потребител	Система
Създаване на сметка от избран тип	Да	Осигурява функционалността
Добавяне на разход или приход към конкретна сметка	Да	Осигурява функционалността
Избор на категория и подкатегория при добавяне на разход или приход	Да	Осигурява функционалността
Избор на конкретна дата при добавяне на разход или приход	Да	Осигурява функционалността
Търсене на определени сметки	Да	Осигурява функционалността
Търсене на определени разходи или приходи	Да	Осигурява функционалността
Филтриране на сметки по тип	Да	Осигурява функционалността
Филтриране на приходи и разходи по категория и подкатегория	Да	Осигурява функционалността
Статистика под формата на диаграми на разходи, приходи и сравнение между двете	Да	Осигурява функционалността
Възможност за филтриране на диаграмите по период	Да	Осигурява функционалността

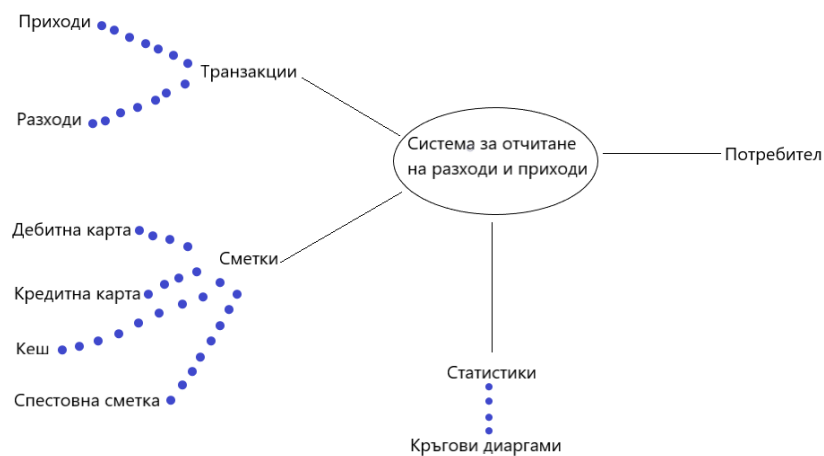
5.5 Карта на съпричастност

Това е инструмент, който ни помага да разберем действителните мисли и актуалното поведение на крайният потребител, като това до голяма степен помага на екипа ни, разработващ системата за отчитане на ежедневни разходи и приходи да навлезе в мислите на групата от потребители, за да бъдем по-информирани и подготвени при разработването на системата.



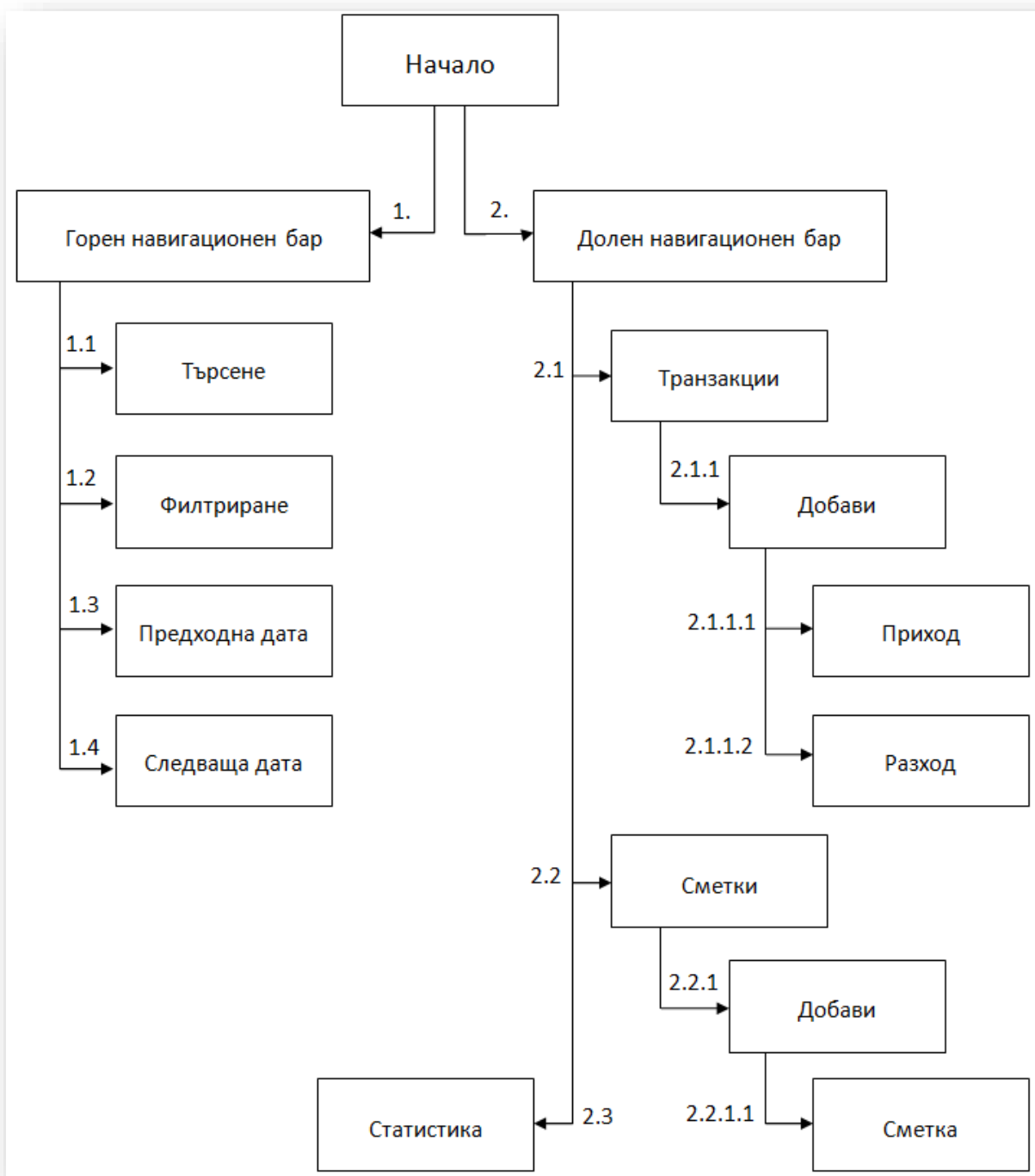
5.6 Mindmap

Мисловната карта това е диаграма използвана за визуално организиране на информация. Тя е йерархична и показва взаимоотношенията между части от цялото. Често се създава около една концепция, нарисувана като изображение в средата, към която се добавят асоциирани изображения на идеи като думи и части от думи. Основните идеи са свързани директно с централната концепция, а другите идеи се разклоняват от тези основни идеи.



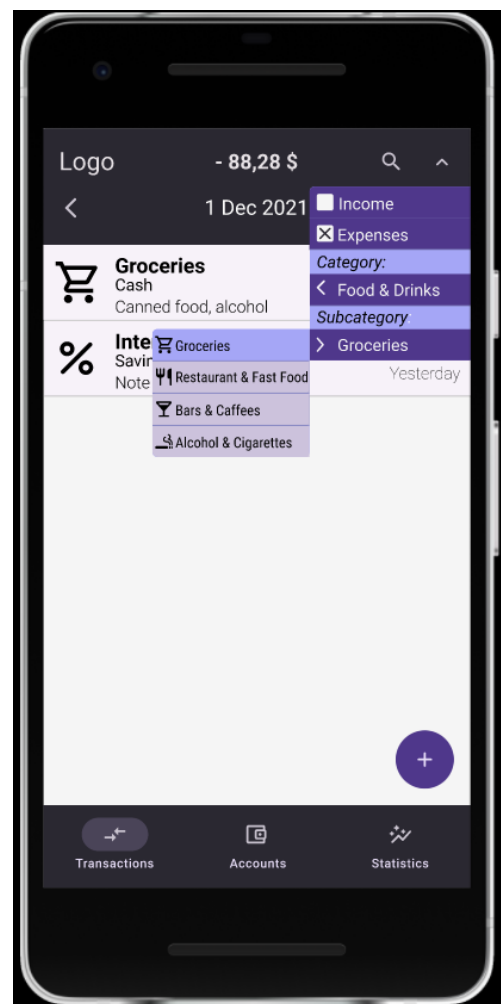
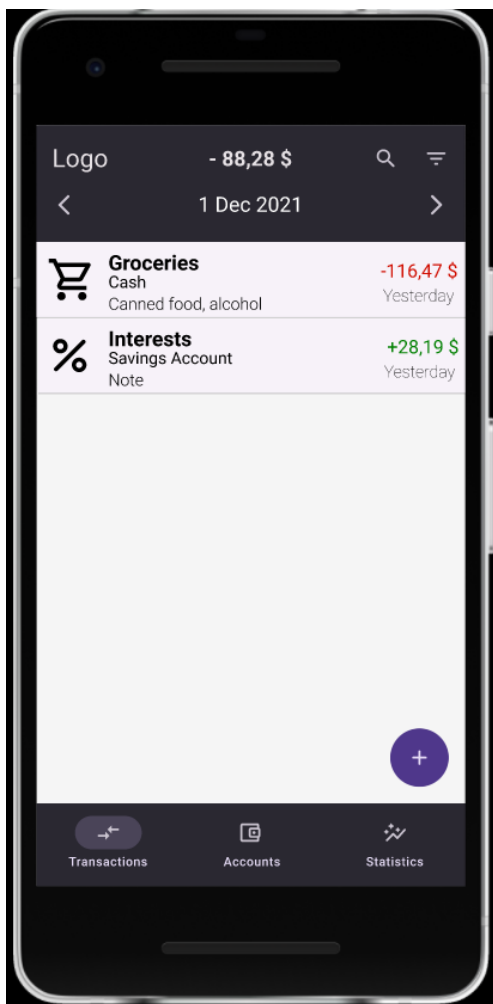
5.7 Sitemap

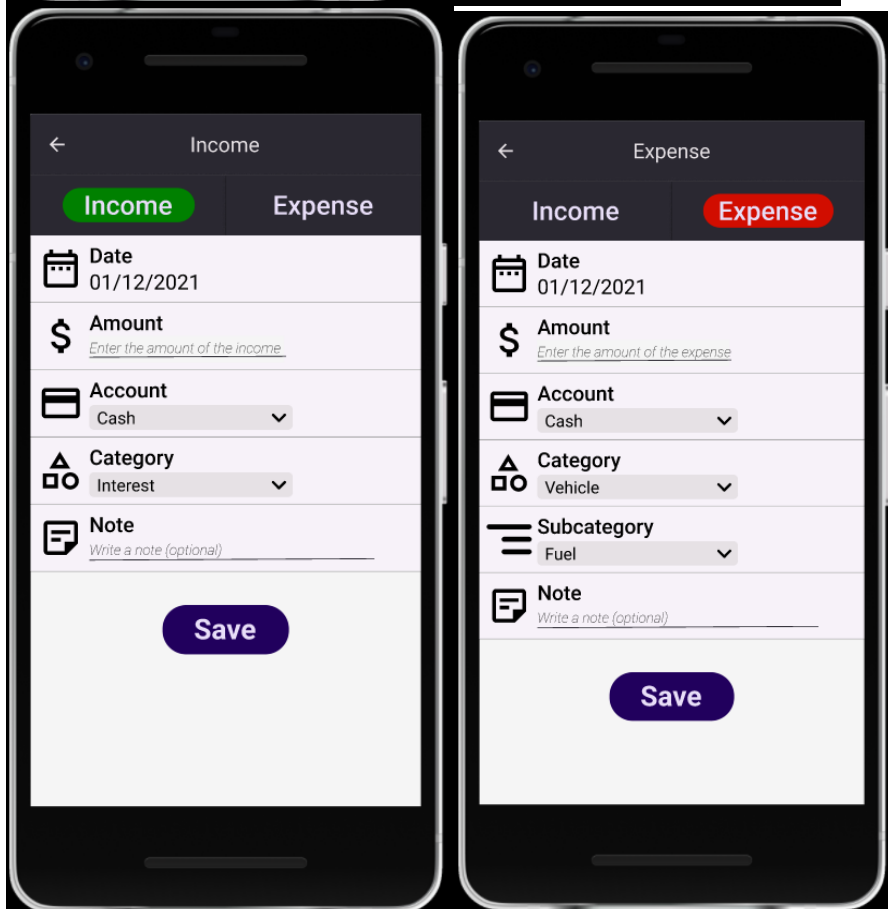
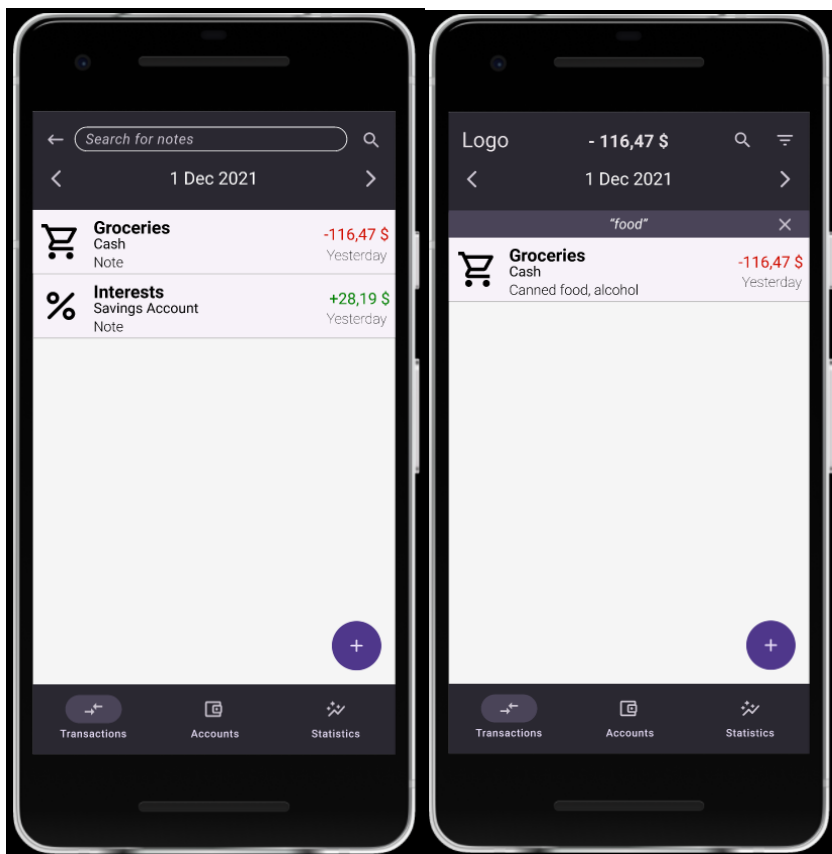
Друг полезен инструмент на етапа на проектиране, който сме използвали е „Sitemap”, чрез който след като имаме готови „Coremodels” и прототип, бихме могли да разработим една дървовидна структура, която пряко да рефлектира или да бъде отражение на нашата система, и да служи като ориентиращ инструмент за навигиране на клиентите в системата.

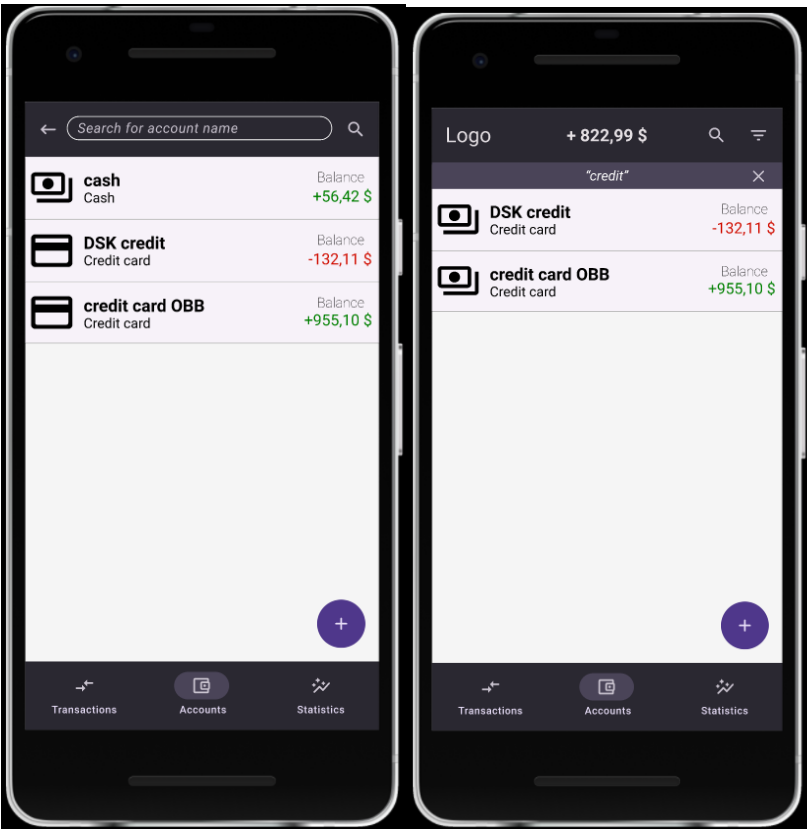
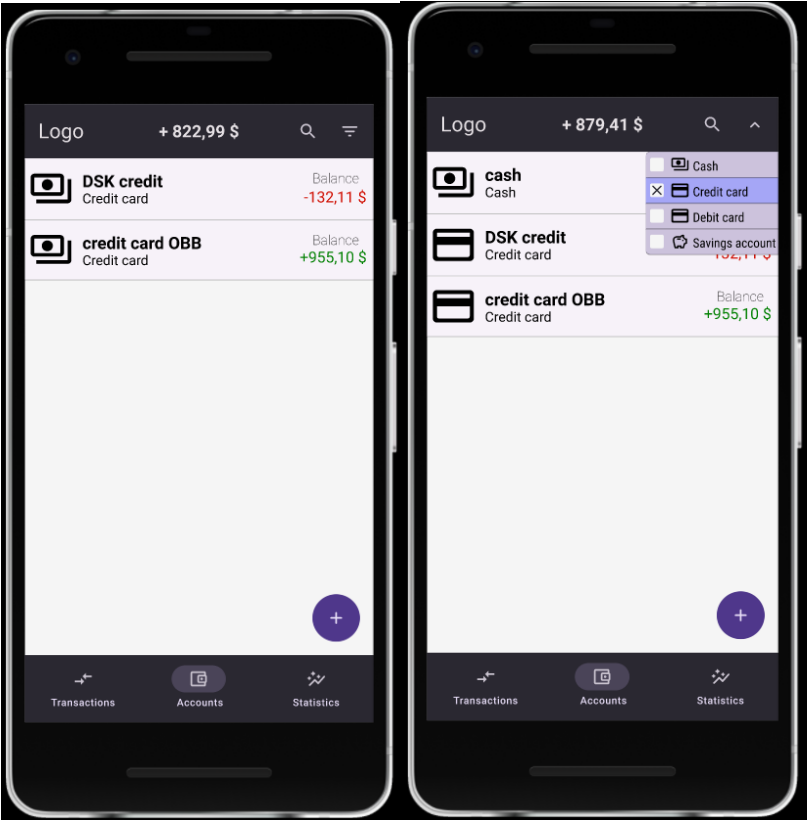


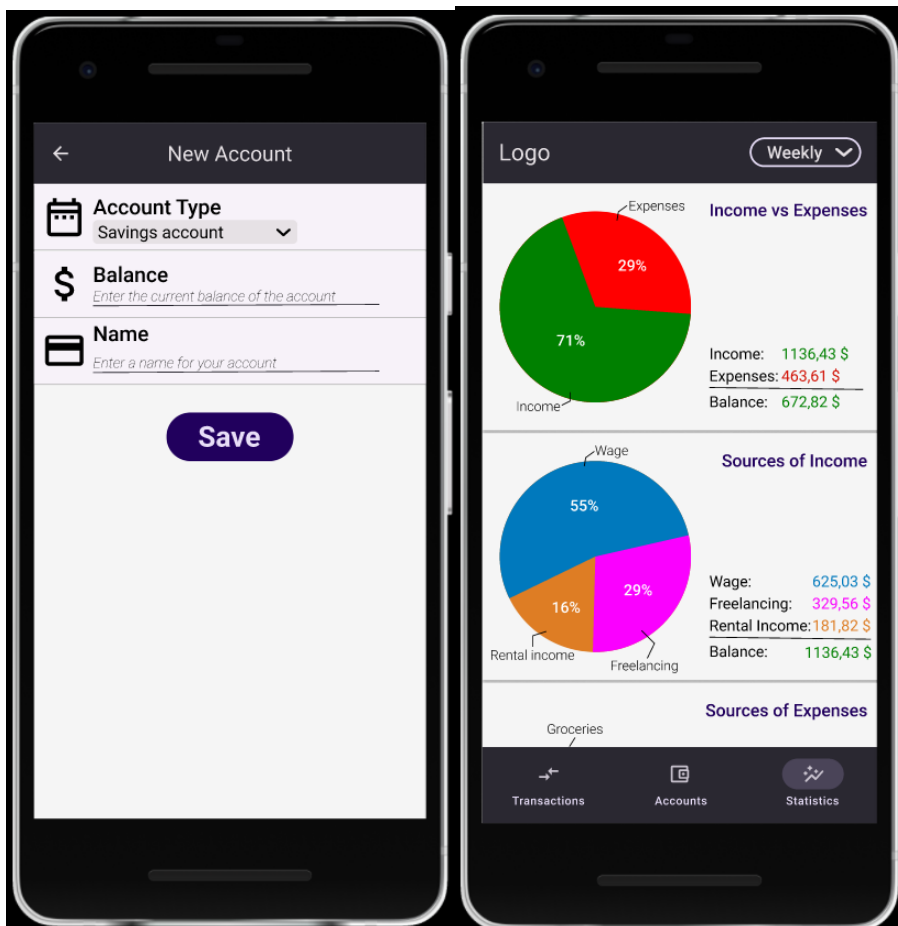
5.8 Прототип

За направата на прототипа ни избрахме да използваме Figma, което е безплатно уеб приложение за изграждане на детайлни прототипи и беше идеално за целта, тъй като поддържа всички елементи на Material Design, наложил се като стандарт за изграждането на UI както на Android, така и на уеб приложения. От хилядите иконки до елементи като Bottom Navigation Bar, Figma позволи изграждането на реалистичен прототип, не особено различаващ се от завършеното ни приложение.



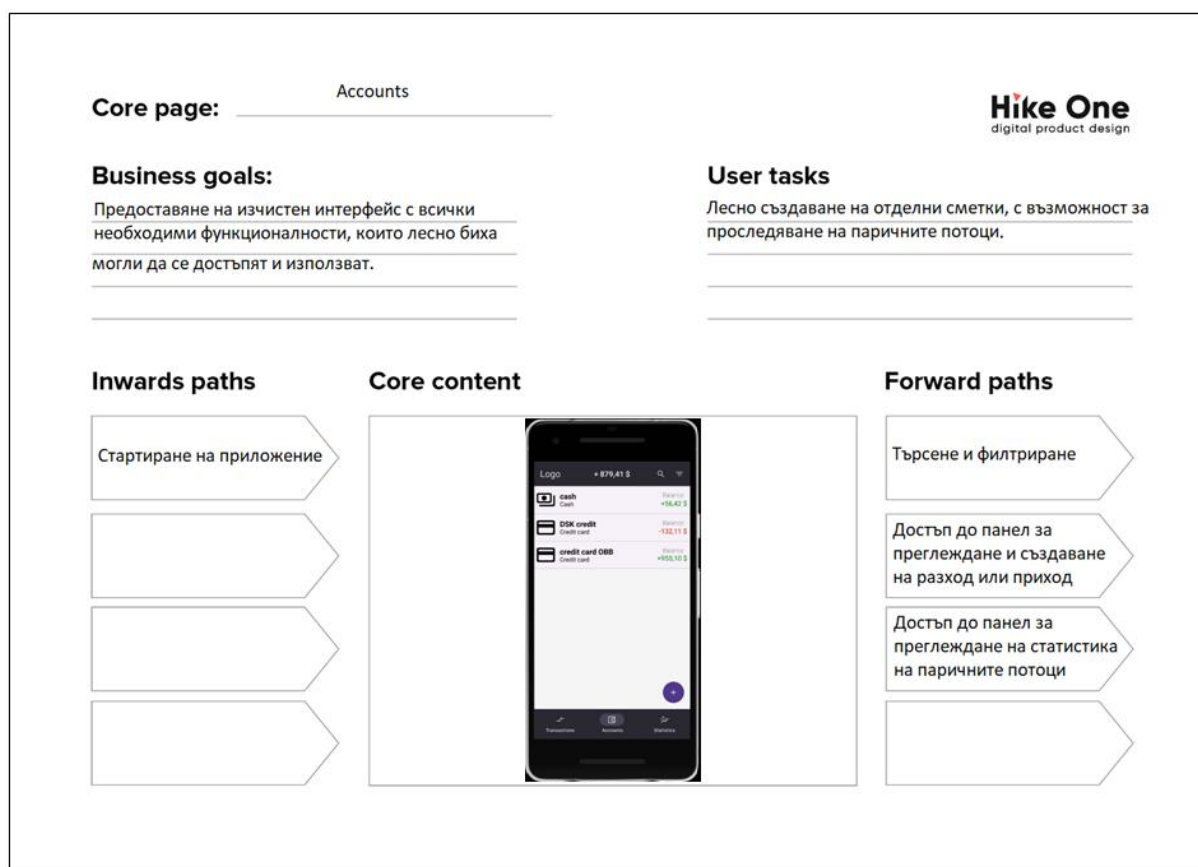






5.9 Coremodels

Следващата стъпка от проектиране на системата, е разработването на така наречените „Coremodels”, чрез които нахвърляме от лявата страна бизнес целта на този модул, пътищата, които водят до него, а от другата страна съответно пътищата до които, той, ще ни отведе както и информация за целта на потребителя, която бихме искали да удовлетворим. Като по средата се представя визуално как би изглеждал прототипа на нашата система. Този инструмент е доста полезен при бъдещо изграждане на прототип, а в следствие и от полза за изграждане на „Sitemap”, за което ще поговорим по-късно. В нашия случай, системата ни има 3 основни визуални части или табове, като това са съответно за Сметки, Транзакции и Статистика. Като главният или „Main” таб е този за сметките на потребителите.



Core page: Transactions

Hike One
digital product design

Business goals:

Предоставяне на изчистен интерфейс с всички необходими функционалности, които лесно биха могли да се достъпят и използват.

User tasks

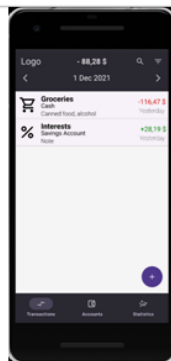
Лесно създаване на разход или приход, с възможност за преглеждане на паричните потоци за конкретна дата.

Inwards paths

Пренасочване от
Accounts

Пренасочване от
Statistics

Core content



Forward paths

Достъп до панел за
преглеждане и създаване
на сметка от избран тип

Достъп до панел за
преглеждане на статистика
на паричните потоци

Търсене и филтриране

Core page: Statistics

Hike One
digital product design

Business goals:

Предоставяне на структурирана статистика под формата на кръгова диаграма за проследяване на разходите и приходите.

User tasks

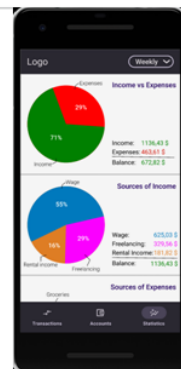
Лесно и бързо преглеждане на статистика на приходите и разходите.

Inwards paths

Пренасочване от
Accounts

Пренасочване от
Transactions

Core content



Forward paths

Достъп до панел за
преглеждане и създаване
на сметка от избран тип

Достъп до панел за
преглеждане и създаване
на разход или приход

Филтриране

6. Фаза на проектиране

6.1 Moodboard

Това е инструмент, чрез който искаме да представим визуално един от проблемите, с които често се сблъскваме, а именно процеса на управление на нашите финанси. Независимо дали се занимаваме професионално във финансовата сфера или просто изчисляваме нашите лични финанси, всеки се сблъсква с този проблем, като бихме искали да разработим и представим една автоматизирана система, която да бъде от ключова полза на крайните потребители и да решава по един ефективен начин този проблем. Именно, чрез мобилно приложение, което всеки би могъл да използва от телефона си по всяко време и да разполага с актуална и навременна информация за своите финансови средства. Като управлението на средствата ще бъде лесно и най-вече продуктивно и ефективно.

Чрез този „Moodboard” също така представяме именованиято на нашето мобилно приложение, както и цветовата гама, която предпочитаме да използваме за графичния интерфейс.



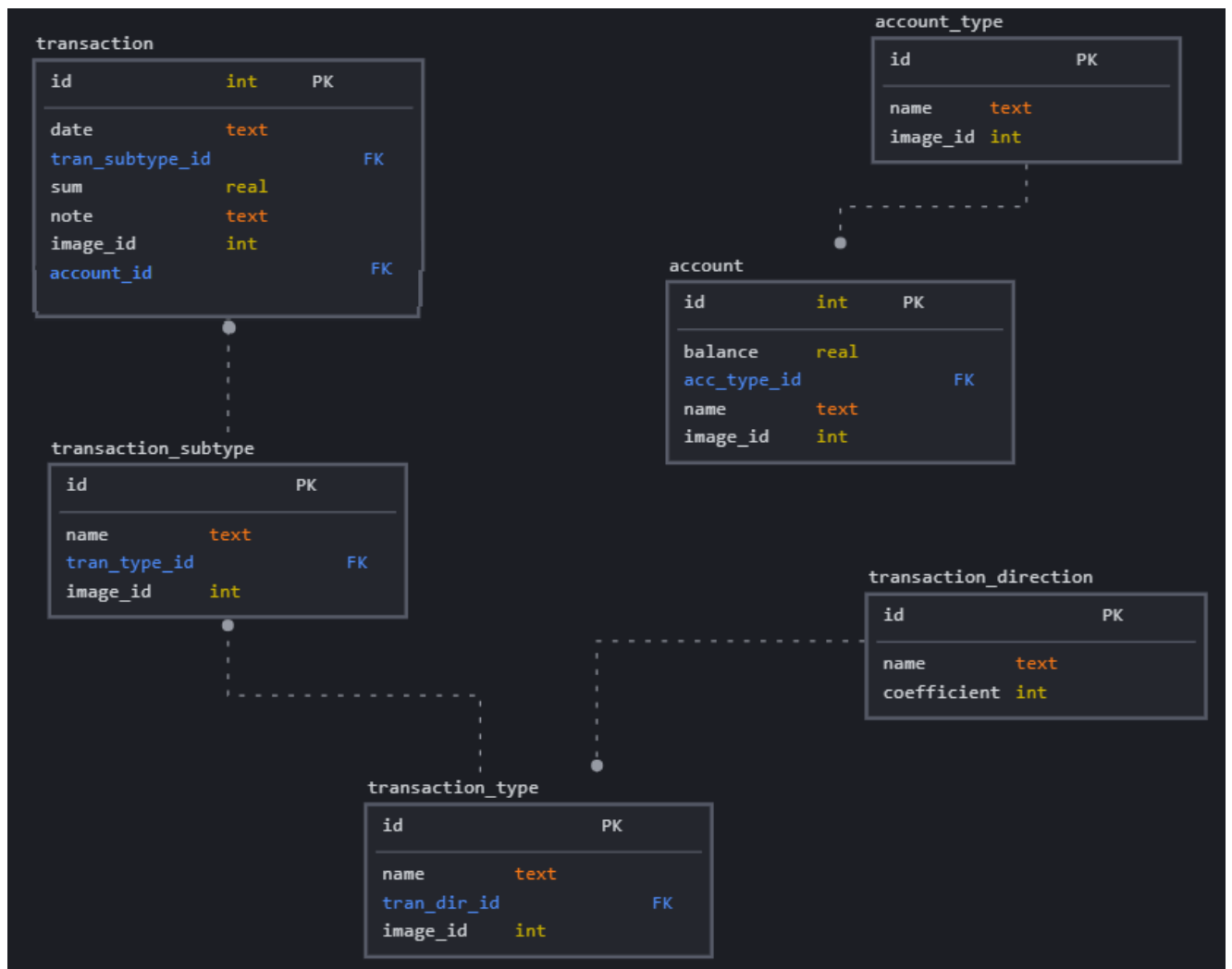
6.2 Лого

Чрез своето лого бихме искали да заявим своята идентичност и лице на нашия дигитален софтуерен продукт, тъй като в самото начало, когато нашия продукт не е познат на потребители, логото е средството, чрез което заявяваме кои сме и какво разработваме. Изборът на подходящи изразни средства като форми, цветове и шрифт помагат да се постигне търсеното внушение. Логото може да бъде средство, чрез което продукта ни да се отличи от своите конкуренти и да демонстрира своята уникалност. Използвайки различно графично решение за своя знак, може да се акцентира върху преимущество, ценност или специално отношение, поради които клиентът да избере нашия продукт пред останалите. Изборът на различен цвят, форма, дори нетипичен за бранша символ (ако разбира се това е добре обосновано), допринася за това, и да бъдат запомнени по-лесно. Предварителното проучване и анализ на логата на конкурентите е задължителна стъпка в дизайна на лого.

Това е нашето лого, идентичност и заявката ни за качествен продукт, на който крайните потребители биха могли да се доверят и използват.



6.3 Модел на базата данни



Описание на таблиците:

account_type: Модел на видовете парични сметки. Съдържа уникален идентификатор, название и идентификатор на иконка.

account: Модел на паричните сметки. Съдържа уникален идентификатор, баланс, идентификатор на Вид Парична Сметка (Account Type), название и идентификатор на иконка.

transaction_direction: Посока на паричните движения. Съдържа уникален идентификатор, название и коефициент: -1 за разходи и 1 за приходи.

transaction_type: Категория на паричното движение. Съдържа уникален идентификатор, идентификатор за Посока на Движението (transaction_direction), название и идентификатор на иконка.

transaction_subtype : Подкатегория на паричното движение. Съдържа уникален идентификатор, идентификатор на Категория на Движението (transaction_type), название и идентификатор на иконка.

transaction: Парично движение. Съдържа уникален идентификатор, идентификатор за Подкатегория на Движението (transaction_subtype), идентификатор за Парична Сметка (account), сума на движението, дата, бележка и идентификатор на иконка.

7. Заключение

Нуждата от правилно управление на личните финанси и повишаване на продуктивността и ефективността, са ключови фактори, поради които се зароди идеята и целта да се проектира и реализира мобилно приложение за управление на финансовите средства. С разработването на тази система се надяваме да улесним ежедневието на крайните потребители и да успеем да предоставим решение на един от техните проблеми, с които се сблъскват всеки ден. Нуждата от управление на паричните средства, никога не е била толкова сериозна, както в днешните времена, и правилната стратегия и подходи откъм инвестиции на средствата в бъдеще са от ключово значение, за нашия живот. Разработването на тази система и предоставянето и на крайните потребители е първата стъпка към решаването на един от многото проблеми в ежедневието ни и предоставяне на нужната ефективност, продуктивност, надеждност и сигурност, при управление на финансовите ни средства.

8. Приложение

8.1 Entity

За да работи Room с обект, трябва да дадете информация на Room, която свързва съдържанието на Java класа на обекта (например AccountsEntity) с това, което искате да представите в таблицата на базата данни. Правите това с помощта на анотации.

Това са някои често използвани анотации:

- `@Entity(tableName = "account")` - Всеки екземпляр `@Entity` представлява обект в таблица. Посочете името на таблицата, ако искате то да е различно от името на класа.
- `@PrimaryKey` - Всеки обект се нуждае от първичен ключ. За да генерирате автоматично уникален ключ за всеки обект, добавете и аотирайте първичен целочислен ключ с `autoGenerate=true`, както е показано в кода по-долу.
- `@NonNull` - Означава, че връщаната стойност на параметър, поле или метод никога не може да бъде Null. Първичният ключ винаги трябва да използва анотацията `@NonNull`. Използвайте тази анотация за задължителни полета във вашите редове.
- `@ColumnInfo(name = "account_name ")` - Посочете името на колоната в таблицата, ако искате името на колоната да е различно от името на променливата-член.

Всяко поле, което се съхранява в базата данни, трябва или да е публично, или да има метод за получаване, за да може Room да има достъп до него. Кодът по-долу предоставя метод `getWord()` "getter", вместо да излага директно променливи-членове.

Това е аотиран код, представляващ Accounts Entity:

```
@Entity(tableName = "account",
    foreignKeys = @ForeignKey(entity = AccountType.class, parentColumns =
    "id", childColumns = "acc_type_id", onDelete = CASCADE, onUpdate = CASCADE),
```



```

        indices = { @Index(name = "Account_PK", value = "id"),
                    @Index(name = "Account_FK", value = "acc_type_id")} )
public class Account implements Parcelable { // Parcelable? Serializable?

    @PrimaryKey(autoGenerate = true)
    private int id;

    private double balance;

    private String name;

    @ColumnInfo(name = "acc_type_id")
    private int accountId;

    // Currency ID zasega lipsva, zashtoto shte dobavi dopolnitelna slojnost
    ///////////////////////////////////

    public Account(double balance, String name, int accountId) {
        this.balance = balance;
        this.name = name;
        this.accountId = accountId;
    }

    protected Account(Parcel in) {
        id = in.readInt();
        balance = in.readDouble();
        name = in.readString();
        accountId = in.readInt();
    }

    public static final Creator<Account> CREATOR = new Creator<Account>() {
        @Override
        public Account createFromParcel(Parcel in) {
            return new Account(in);
        }

        @Override
        public Account[] newArray(int size) {
            return new Account[size];
        }
    };

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public double getBalance() {
        return balance;
    }

    public String getName() {
        return name;
    }
}

```

```
public void setName(String name) {
    this.name = name;
}

public void setBalance(double balance) {
    this.balance = balance;
}

public int getAccountTypeId() {
    return accountTypeId;
}

public void setAccountTypeId(int accountTypeId) {
    this.accountTypeId = accountTypeId;
}

@Override
public int describeContents() {
    return 0;
}

@Override
public void writeToParcel(Parcel dest, int flags) {
    dest.writeInt(id);
    dest.writeDouble(balance);
    dest.writeString(name);
    dest.writeInt(accountTypeId);
}
```

8.2 DAO (Data Access Object)

За да получите достъп до данните на приложението си чрез библиотеката за постоянство на стаята, работите с обекти за достъп до данни или DAO. Набор от Dao обекти (DAOs) формира основния компонент на Room database. Всеки DAO включва методи, които предлагат абстрактен достъп до базата данни на вашето приложение. Анотирате DAO, за да укажете SQL заявки и да ги свържете с извиквания на методи. Компиляторът проверява SQL за грешки, след което генерира заявки от анотациите. За често срещани заявки библиотеките предоставят удобни анотации, като @Insert, @Delete и @Update.

Това е код, който демонстрира тези пояснения заедно с различни типове заявки:

```
@Dao
public interface AccountTypeDao {

    @Transaction
    @Insert
    long insertAccountType(AccountType accountType);

    @Transaction
    @Insert
    long insertAccount(Account account);

    @Transaction
    @Update
    void updateAccountType(AccountType accountType);

    @Transaction
    @Update
    void updateAccount(Account account);

    @Transaction
    @Delete
    void deleteAccountType(AccountType accountType);

    @Transaction
    @Delete
    void deleteAccount(Account account);

    @Transaction
    @Query("DELETE FROM account_type")
    void deleteAllAccountTypes();

    @Transaction
    @Query("DELETE FROM account")
    void deleteAllAccounts();

    //////////////////////////////////////
    //////////////////////////////////////
    // SELECT & SELECT WHERE id
    // for:
    // - AccountType (with Accounts ||| RELATIONSHIP)
    // - Account (SIMPLE ||| NO RELATIONSHIP) ---> for relationship with
    Transactions see *TransactionDao*
```

```
@Transaction
@Query("SELECT * FROM account_type")
LiveData<List<AccountType>> getAllAccountTypes();

@Transaction
@Query("SELECT * FROM account_type WHERE id = :arg0")
LiveData<AccountType> getAccountTypeById(int arg0);

@Transaction
@Query("SELECT * FROM account")
LiveData<List<Account>> getAllAccounts(); // N

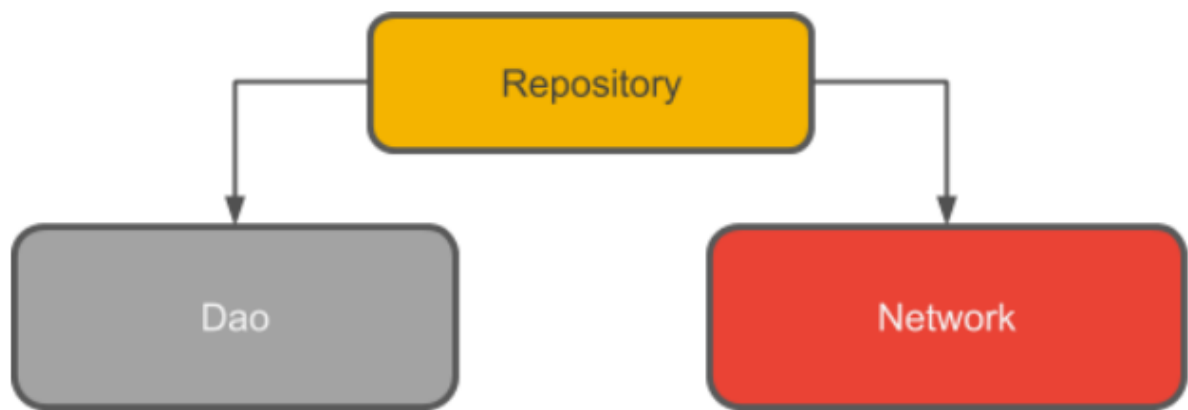
@Transaction
@Query("SELECT * FROM account WHERE id = :arg0")
LiveData<Account> getAccountById(int arg0);

@Transaction
@Query("SELECT * FROM account_type")
LiveData<List<AccountTypeWithAccounts>> getAllAccountTypesWithAccounts();

@Transaction
@Query("SELECT * FROM account_type where id = :id")
LiveData<AccountTypeWithAccounts> getAccountTypeWithAccountsById(int id);
}
```

8.3 Repository

Хранилището е клас, който абстрахира достъпа до множество източници на данни. Хранилището не е част от библиотеките на компонентите на архитектурата, но е препоръчителна най-добра практика за разделяне на кода от архитектурата. Клас Repository обработва операции с данни. Той предоставя чист API за останалата част от приложението за работа с база данни. Хранилището е мястото, където бихте поставили кода, за да управлявате нишките на заявки и да използвате множество бекендове, ако е подходящо. Обичайната употреба за хранилище е да се внедри логиката за вземане на решение дали да се извличат данни от мрежа или да се използват резултати, кеширани в базата данни.



Това е примерен код за основно хранилище:

```
public class AccountTypeRepository {  
    private AccountTypeDao oAccountTypeDao;  
    private LiveData<List<Account>> oLiveDataListAllAccounts;  
    private LiveData<List<AccountType>> oLiveDataListAllAccountTypes;  
    private LiveData<List<AccountTypeWithAccounts>>  
oLiveDataListAccountTypesWithAccounts;  
  
    public AccountTypeRepository(Application application) {  
        AppDatabase oAppDatabase = AppDatabase.getInstance(application);  
  
        oAccountTypeDao = oAppDatabase.accountTypeDao();  
  
        oLiveDataListAllAccounts = oAccountTypeDao.getAllAccounts();  
        oLiveDataListAllAccountTypes = oAccountTypeDao.getAllAccountTypes();  
        oLiveDataListAccountTypesWithAccounts =  
oAccountTypeDao.getAllAccountTypesWithAccounts();  
    }  
  
    public LiveData<List<AccountTypeWithAccounts>>
```

```

getAllAccountTypesWithAccounts() {
    return oLiveDataListAccountTypesWithAccounts;
}

public LiveData<Account> getAccountByID(int nID){
    new getAccountByIDAsyncTask(oAccountTypeDao).execute(nID);
    return getAccountByIDAsyncTask.getAccount();
}

public int insertAccount(Account oAccount){
    new InsertAccountAsyncTask(oAccountTypeDao).execute(oAccount);
    return InsertAccountAsyncTask.getAccountID();
}

public void updateAccount(Account oAccount){
    new UpdateAccountAsyncTask(oAccountTypeDao).execute(oAccount);
}

public void deleteAccount(Account oAccount){
    new DeleteAccountAsyncTask(oAccountTypeDao).execute(oAccount);
}

public void deleteAllAccounts(){
    new DeleteAllAccountsAsyncTask(oAccountTypeDao).execute();
}

public LiveData<List<Account>> getAllAccounts() {
    return oLiveDataListAllAccounts;
}

public LiveData<AccountType> getAccountTypeByID(int nID){
    new getAccountTypeByIDAsyncTask(oAccountTypeDao).execute(nID);
    return getAccountTypeByIDAsyncTask.getAccountType();
}

public int insertAccountType(AccountType oAccountType){
    new InsertAccountTypeAsyncTask(oAccountTypeDao).execute(oAccountType);
    return InsertAccountTypeAsyncTask.getAccountTypeID();
}

public void updateAccountType(AccountType oAccountType){
    new UpdateAccountTypeAsyncTask(oAccountTypeDao).execute(oAccountType);
}

public void deleteAccountType(AccountType oAccountType){
    new DeleteAccountTypeAsyncTask(oAccountTypeDao).execute(oAccountType);
}

public void deleteAllAccountTypes(){
    new DeleteAllAccountTypesAsyncTask(oAccountTypeDao).execute();
}

public LiveData<List<AccountType>> getAllAccountTypes() {
    return oLiveDataListAllAccountTypes;
}

private static class getAccountByIDAsyncTask extends AsyncTask<Integer, Void,
LiveData<Account>> {
    private AccountTypeDao oAccountTypeDao;

```

```

        private static LiveData<Account> oAccount;

        private getAccountByIdAsyncTask(AccountTypeDao oAccountTypeDao){
            this.oAccountTypeDao = oAccountTypeDao;
        }

        @Override
        protected LiveData<Account> doInBackground(Integer... integers) {
            oAccount = oAccountTypeDao.getAccountById(integers[0]);
            return oAccountTypeDao.getAccountById(integers[0]);
        }

        @Override
        protected void onPostExecute(LiveData<Account> oAccount) {
            this.oAccount = oAccount;
        }

        public static LiveData<Account> getAccount() {
            return oAccount;
        }
    }

    private static class InsertAccountAsyncTask extends AsyncTask<Account, Void,
Long>{
        private AccountTypeDao oAccountTypeDao;
        private static int nAccountID;

        private InsertAccountAsyncTask(AccountTypeDao oAccountTypeDao){
            this.oAccountTypeDao = oAccountTypeDao;
        }

        @Override
        protected Long doInBackground(Account... accounts) {
            return oAccountTypeDao.insertAccount(accounts[0]);
        }

        @Override
        protected void onPostExecute(Long aLong) {
            this.nAccountID = aLong.intValue();
        }

        public static int getAccountID() {
            return nAccountID;
        }
    }

    private static class UpdateAccountAsyncTask extends AsyncTask<Account, Void,
Void>{
        private AccountTypeDao oAccountTypeDao;

        private UpdateAccountAsyncTask(AccountTypeDao oAccountTypeDao){
            this.oAccountTypeDao = oAccountTypeDao;
        }

        @Override
        protected Void doInBackground(Account... accounts) {
            oAccountTypeDao.updateAccount(accounts[0]);
            return null;
        }
    }

```

```

    }

    private static class DeleteAccountAsyncTask extends AsyncTask<Account, Void,
Void>{
        private AccountTypeDao oAccountTypeDao;

        private DeleteAccountAsyncTask(AccountTypeDao oAccountTypeDao){
            this.oAccountTypeDao = oAccountTypeDao;
        }

        @Override
        protected Void doInBackground(Account... accounts) {
            oAccountTypeDao.deleteAccount(accounts[0]);
            return null;
        }
    }

    private static class DeleteAllAccountsAsyncTask extends AsyncTask<Void, Void,
Void>{
        private AccountTypeDao oAccountTypeDao;

        private DeleteAllAccountsAsyncTask(AccountTypeDao oAccountTypeDao){
            this.oAccountTypeDao = oAccountTypeDao;
        }

        @Override
        protected Void doInBackground(Void... voids) {
            oAccountTypeDao.deleteAllAccounts();
            return null;
        }
    }

    private static class getAccountTypeByIDAsyncTask extends AsyncTask<Integer,
Void, LiveData<AccountType>> {
        private AccountTypeDao oAccountTypeDao;
        private static LiveData<AccountType> oAccountType;

        private getAccountTypeByIDAsyncTask(AccountTypeDao oAccountTypeDao){
            this.oAccountTypeDao = oAccountTypeDao;
        }

        @Override
        protected LiveData<AccountType> doInBackground(Integer... integers) {
            return oAccountTypeDao.getAccountTypeByID(integers[0]);
        }

        @Override
        protected void onPostExecute(LiveData<AccountType> oAccountType) {
            this.oAccountType = oAccountType;
        }

        public static LiveData<AccountType> getAccountType() {
            return oAccountType;
        }
    }

    private static class InsertAccountTypeAsyncTask extends AsyncTask<AccountType,
Void, Long>{
        private AccountTypeDao oAccountTypeDao;

```



```

        private static int nAccountTypeID;

        private InsertAccountTypeAsyncTask(AccountTypeDao oAccountTypeDao){
            this.oAccountTypeDao = oAccountTypeDao;
        }

        @Override
        protected Long doInBackground(AccountType... accountTypes) {
            return oAccountTypeDao.insertAccountType(accountTypes[0]);
        }

        @Override
        protected void onPostExecute(Long aLong) {
            this.nAccountTypeID = aLong.intValue();
        }

        public static int getAccountTypeID() {
            return nAccountTypeID;
        }
    }

    private static class UpdateAccountTypeAsyncTask extends AsyncTask<AccountType,
Void, Void>{
        private AccountTypeDao oAccountTypeDao;

        private UpdateAccountTypeAsyncTask(AccountTypeDao oAccountTypeDao){
            this.oAccountTypeDao = oAccountTypeDao;
        }

        @Override
        protected Void doInBackground(AccountType... accountTypes) {
            oAccountTypeDao.updateAccountType(accountTypes[0]);
            return null;
        }
    }

    private static class DeleteAccountTypeAsyncTask extends AsyncTask<AccountType,
Void, Void>{
        private AccountTypeDao oAccountTypeDao;

        private DeleteAccountTypeAsyncTask(AccountTypeDao oAccountTypeDao){
            this.oAccountTypeDao = oAccountTypeDao;
        }

        @Override
        protected Void doInBackground(AccountType... accountTypes) {
            oAccountTypeDao.deleteAccountType(accountTypes[0]);
            return null;
        }
    }

    private static class DeleteAllAccountTypesAsyncTask extends AsyncTask<Void,
Void, Void>{
        private AccountTypeDao oAccountTypeDao;

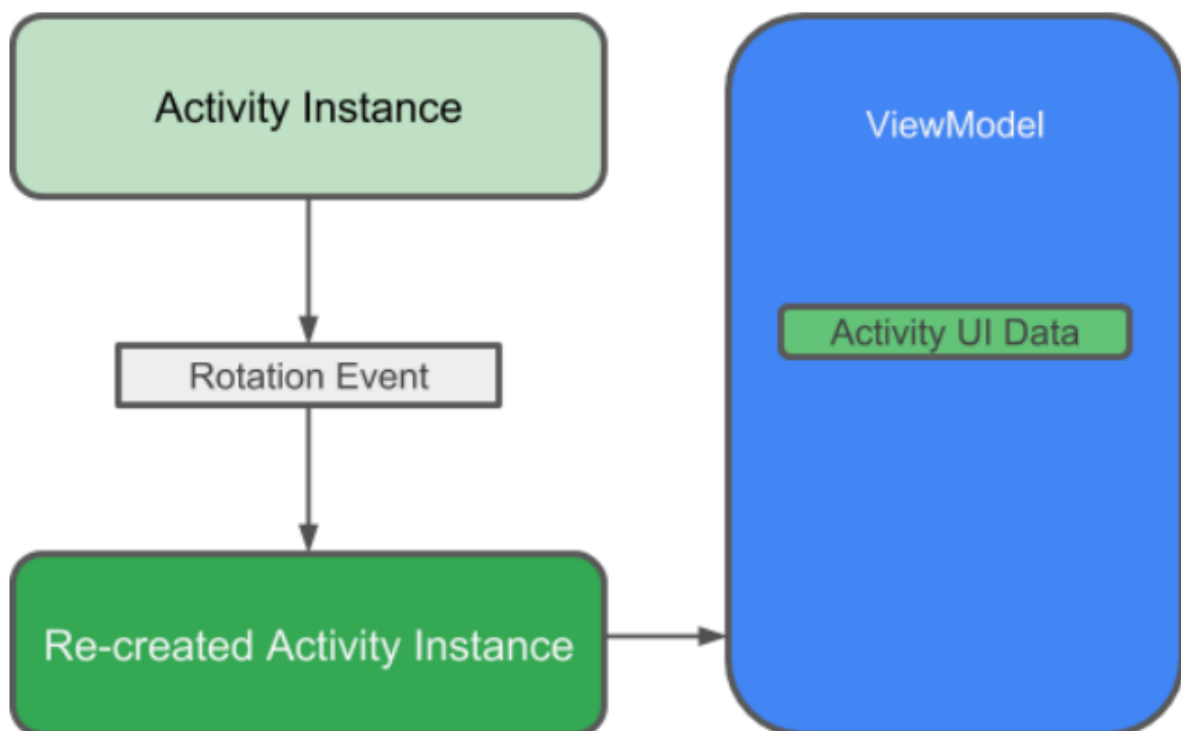
        private DeleteAllAccountTypesAsyncTask(AccountTypeDao oAccountTypeDao){
            this.oAccountTypeDao = oAccountTypeDao;
        }
    }

```

```
    @Override
    protected Void doInBackground(Void... voids) {
        oAccountTypeDao.deleteAllAccountTypes();
        return null;
    }
}
```

8.4 Viewmodel

ViewModel е клас, чиято роля е да предоставя данни на потребителския интерфейс и да оцелее при промени в конфигурацията. ViewModel действа като комуникационен център между хранилището и потребителския интерфейс. Можем също да използваме ViewModel за споделяне на данни между фрагменти. ViewModel е част от библиотеката на жизнения цикъл. ViewModel съхранява данните за потребителския интерфейс на нашето приложение по начин, съобразен с жизнения цикъл, който оцелява при промени в конфигурацията. Разделянето на данните за потребителския интерфейс на нашето приложение от нашите класове Activity и Fragment ни позволява по-добре да следваме принципа на единна отговорност: нашите дейности и фрагменти са отговорни за изтеглянето на данни на екрана, докато нашият ViewModel е отговорен за съхраняването и обработката на всички данни, необходими за потребителския интерфейс.



Това е примерен код за Viewmodel:

```
public class AccountTypeViewModel extends AndroidViewModel {

    private AccountTypeRepository oAccountTypeRepository;
    private LiveData<List<Account>> oLiveDataListAllAccounts;
    private LiveData<List<AccountType>> oLiveDataListAllAccountTypes;
    private LiveData<List<AccountTypeWithAccounts>>
oLiveDataListAccountTypesWithAccounts;

    public AccountTypeViewModel(@NonNull Application application) {
        super(application);
        oAccountTypeRepository = new AccountTypeRepository(application);
        oLiveDataListAllAccounts = oAccountTypeRepository.getAllAccounts();
        oLiveDataListAllAccountTypes =
oAccountTypeRepository.getAllAccountTypes();
        oLiveDataListAccountTypesWithAccounts =
oAccountTypeRepository.getAllAccountTypesWithAccounts();
    }

    public LiveData<AccountType> getAccountTypeByID(int nID){
        return oAccountTypeRepository.getAccountTypeByID(nID);
    }

    public int insertAccountType(AccountType oAccountType){
        return oAccountTypeRepository.insertAccountType(oAccountType);
    }

    public void updateAccountType(AccountType oAccountType){
        oAccountTypeRepository.updateAccountType(oAccountType);
    }

    public void deleteAccountType(AccountType oAccountType){
        oAccountTypeRepository.deleteAccountType(oAccountType);
    }

    public void deleteAllAccountTypes(){
        oAccountTypeRepository.deleteAllAccountTypes();
    }

    public LiveData<List<AccountType>> getAllAccountTypes(){
        return oLiveDataListAllAccountTypes;
    }

    public LiveData<Account> getAccountByID(int nID){
        return oAccountTypeRepository.getAccountByID(nID);
    }

    public int insertAccount(Account oAccount){
        return oAccountTypeRepository.insertAccount(oAccount);
    }

    public void updateAccount(Account oAccount){
        oAccountTypeRepository.updateAccount(oAccount);
    }

    public void deleteAccount(Account oAccount){
```

```
        oAccountTypeRepository.deleteAccount(oAccount);
    }

    public void deleteAllAccounts(){
        oAccountTypeRepository.deleteAllAccounts();
    }

    public LiveData<List<Account>> getAllAccounts(){
        return oLiveDataListAllAccounts;
    }

    public LiveData<List<AccountTypeWithAccounts>>
getAllAccountTypesWithAccounts(){
        return oLiveDataListAccountTypesWithAccounts;
    }
}
```

8.5 LiveData

Всеки път, когато данните се променят, методът `onChanged()` на нашия наблюдател се извиква. В най-основния случай това може да актуализира съдържанието на `TextView`, както е показано в този код:

```
final Observer<String> nameObserver = new Observer<String>() {  
    @Override  
    public void onChanged(@Nullable final String newName) {  
        // Update the UI, in this case, a TextView.  
        mNameTextView.setText(newName);  
    }  
};
```

Друг често срещан случай е показването на данни в изглед, който работи с адаптер. Например, ако показваме данни в `RecyclerView`, методът `onChanged()` актуализира данните, кеширани в адаптера:

```
oViewModel.getAllAccountTypesWithAccounts().observe(getViewLifecycleOwner(), new  
Observer<List<AccountTypeWithAccounts>>() {  
    @Override  
    public void onChanged(@Nullable final List<AccountTypeWithAccounts>  
accTypesWithAccounts) {  
        // Update the cached copy of the words in the adapter.  
        adapter.setData(accTypesWithAccounts);  
    }  
});
```