** МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ**

**РОССИЙСКОЙ ФЕДЕРАЦИИ**

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ**

**УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**

**«МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ**

(национальный исследовательский университет)»

**Институт (Филиал)** № 8 «Компьютерные науки и прикладная математика» **Кафедра**  806

**Группа** М8О-411Б-20 **Направление подготовки** 02.03.02 Фундаментальная информатика и информационные технологии

**Профиль**  Информатика и компьютерные науки

**Квалификация**  **бакалавр**

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА**

**БАКАЛАВРА**

На тему: Программная реализация приложения “Хранилище паспортов ископаемых музейных экспонатов”

Автор ВКРБ\_\_\_\_\_\_\_\_\_\_\_\_\_\_Ким Илья Васильевич (\_\_\_\_\_\_\_\_\_\_)

(фамилия, имя, отчество полностью)

Руководитель Лемтюжникова Дарьяна Владимировна (\_\_\_\_\_\_\_\_\_\_)

(фамилия, имя, отчество полностью)

Консультант (\_\_\_\_\_\_\_\_\_\_)

(фамилия, имя, отчество полностью)

Консультант (\_\_\_\_\_\_\_\_\_\_)

(фамилия, имя, отчество полностью)

Рецензент (\_\_\_\_\_\_\_\_\_\_)

(фамилия, имя, отчество полностью)

**К защите допустить**

Заведующий кафедрой 806 Крылов Сергей Сергеевич (\_\_\_\_\_\_\_\_\_)

(№ каф) (фамилия, имя, отчество полностью)

\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2024г.

Москва 2024

**РЕФЕРАТ**

(TODO: переделать). Выпускная квалификационная работа бакалавра содержит 121 страницу, 61 рисунок, 0 таблиц, 14 использованных источников, 25 приложений.

QR-КОД, ИНВЕНТАРИЗАЦИЯ, MVVM, АРХИТЕКТУРА ПО, ANDROID, SPRING ФРЕЙМВОРК, БАЗЫ ДАННЫХ, МОБИЛЬНАЯ РАЗРАБОТКА.

Объектом исследования являются архитектурные и программные решения создания мобильного приложения для хранения и работой с паспортами музейных экспонатов с применением актуальных подходов и технологий в области разработки мобильных приложений и архитектуры соответствующей базы данных.

Целью данной работы является создание многофункционального, удобного программного комплекса, который позволяет пользователю свободно работать с информацией об объектах музейных экспонатов — в частности добавление новых паспортов, удаление и редактирование существующих. (написать про режимы здесь?) и оперировать электронными версиями документов при помощи мобильного приложения.

Для разработки хранилища использовалась база данных PostgreSql, мобильное приложение написано при помощи фреймворка Flutter и языка Dart. Приложение спроектировано на основе паттерна MVC.

В результате создана система, позволяющая пользователю взаимодействовать с информацией об объектах, редактировать данные или удалять их из хранилища (при наличии соответствующих прав) и быстро вносить новые паспорта музейных объектов в хранилище.

Разработанное решение позволит существенно упростить и сделать более информационно емким процесс описания музейных экспонатов в сравнении с традиционным подходом.

**СОДЕРЖАНИЕ**

[ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ 6](#_Toc160710206)

[ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ 7](#_Toc160710207)

[ВВЕДЕНИЕ 9](#_Toc160710208)

[ОСНОВНАЯ ЧАСТЬ 11](#_Toc160710209)

[1 ТЕОРЕТИЧЕСКАЯ ЧАСТЬ 11](#_Toc160710210)

[1.1 Обзор проблематики 11](#_Toc160710211)

[1.2 Технологии для создания серверной части 14](#_Toc160710212)

[1.2.1 Язык и фреймворк 14](#_Toc160710213)

[1.2.2 Базы данных 17](#_Toc160710214)

[1.3 Технологии для создания мобильного приложения 19](#_Toc160710215)

[1.3.1. Операционная система для разработки 19](#_Toc160710220)

[1.3.2. Подходы создания пользовательского интерфейса 20](#_Toc160710221)

[1.4 Сравнение архитектурных шаблонов для Android разработки 24](#_Toc160710222)

[1.4.1 MVVM 25](#_Toc160710223)

[1.4.2 MVI 27](#_Toc160710230)

[1.4.3 MVP 29](#_Toc160710231)

[1.4.4 MVC 30](#_Toc160710232)

[1.4.5 Вывод 31](#_Toc160710242)

[1.5 ПРАКТИЧЕСКАЯ ЧАСТЬ 31](#_Toc160710243)

[1.5.1 Функционал и архитектура приложения 32](#_Toc160710244)

[1.5.2 Модель базы данных 33](#_Toc160710245)

[1.5.3 Выбор инструментов и технологий 35](#_Toc160710246)

[1.5.4 QR-код в приложении 37](#_Toc160710247)

[1.5.5 Серверная часть 39](#_Toc160710248)

[1.5.5.1 Репозитории 39](#_Toc160710249)

[1.5.5.2 Сервисы 41](#_Toc160710250)

[1.5.5.3 Контроллеры и endpoints 41](#_Toc160710251)

[1.5.5.4 Mapper 43](#_Toc160710252)

[1.5.5.5 Компонент перехвата исключительных ситуаций 45](#_Toc160710253)

[1.5.5.6 Система проверки API Swagger 45](#_Toc160710254)

[1.5.5.7 Вывод 46](#_Toc160710255)

[1.5.6 Клиентская часть 47](#_Toc160710256)

[1.5.6.1 Реализация шаблона MVVM 47](#_Toc160710257)

[1.5.6.2 Граф навигации экранов 50](#_Toc160710258)

[1.5.6.3 Внедрение зависимостей 51](#_Toc160710259)

[1.5.6.4 Авторизация 52](#_Toc160710260)

[1.5.6.5 Добавление 53](#_Toc160710261)

[1.5.6.6 Сканирование 56](#_Toc160710262)

[1.5.6.7 Модификация 58](#_Toc160710263)

[1.5.6.8 Списание 58](#_Toc160710264)

[1.5.6.9 Хранение 59](#_Toc160710265)

[1.5.6.10 Инвентаризация 62](#_Toc160710266)

[3 РЕЗУЛЬТАТЫ РАБОТЫ 65](#_Toc160710267)

[3.1 Сферы применения программного комплекса 65](#_Toc160710270)

[3.2 Результаты разработки 65](#_Toc160710271)

[3.3 Проведение эксперимента и результаты 65](#_Toc160710272)

[ЗАКЛЮЧЕНИЕ 69](#_Toc160710273)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 71](#_Toc160710274)

[ПРИЛОЖЕНИЯ 73](#_Toc160710275)

[ПРИЛОЖЕНИЕ А 73](#_Toc160710276)

[ПРИЛОЖЕНИЕ Б 74](#_Toc160710277)

[ПРИЛОЖЕНИЕ В 75](#_Toc160710278)

[ПРИЛОЖЕНИЕ Г 76](#_Toc160710279)

[ПРИЛОЖЕНИЕ Д 77](#_Toc160710280)

[ПРИЛОЖЕНИЕ Е 78](#_Toc160710281)

[ПРИЛОЖЕНИЕ Ж 79](#_Toc160710282)

[ПРИЛОЖЕНИЕ И 81](#_Toc160710283)

[ПРИЛОЖЕНИЕ К 84](#_Toc160710284)

[ПРИЛОЖЕНИЕ Л 85](#_Toc160710285)

[ПРИЛОЖЕНИЕ М 88](#_Toc160710286)

[ПРИЛОЖЕНИЕ Н 89](#_Toc160710287)

[ПРИЛОЖЕНИЕ П 92](#_Toc160710288)

[ПРИЛОЖЕНИЕ Р 93](#_Toc160710289)

[ПРИЛОЖЕНИЕ С 99](#_Toc160710290)

[ПРИЛОЖЕНИЕ Т 102](#_Toc160710291)

[ПРИЛОЖЕНИЕ У 104](#_Toc160710292)

[ПРИЛОЖЕНИЕ Ф 105](#_Toc160710293)

[ПРИЛОЖЕНИЕ Х 106](#_Toc160710294)

[ПРИЛОЖЕНИЕ Ц 107](#_Toc160710295)

[ПРИЛОЖЕНИЕ Ш 108](#_Toc160710296)

[ПРИЛОЖЕНИЕ Щ 110](#_Toc160710297)

[ПРИЛОЖЕНИЕ Э 118](#_Toc160710298)

[ПРИЛОЖЕНИЕ Ю 120](#_Toc160710299)

[ПРИЛОЖЕНИЕ Я 121](#_Toc160710300)

# ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ

В настоящей выпускной квалификационной работе бакалавра применяют следующие термины с соответствующими определениями:

Государственный учет – комплекс мер, обеспечивающих идентификацию и предметно-количественный учет музейных предметов и музейных коллекций.

Книга поступлений – основной юридический документ учета, научного описания и охраны памятников истории и культуры музея.

Музейный паспорт – важный документ, содержащий всю необходимую информацию об отдельном музейном предмете или коллекции экспонатов.

Паспортизация – процесс присвоения объектам паспортов с соответствующей информацией.

Фреймворк – (с англ. framework — «каркас, структура») — заготовка, готовая модель в программировании для быстрой разработки. Он задает структуру, определяет правила и предоставляет необходимый набор инструментов для создания проекта.

Реляционная база данных – набор данных с установленными связями между ними.

Изолированностью в ACID – способность совершать транзакции.

Согласованность означает, что транзакция будет выполнена только в том случае, если она не нарушает согласованность данных в базе.

Изолированность системы в ACID означает, что параллельные действия не влияют друг на друга.

Устойчивость системы в ACID означает, что уже выполненная транзакция не отменится из-за каких-либо неполадок, например, потеря электроэнергии

Репликация – создание резервной копии базы данных.

Протокол – набор правил, задающих форматы сообщений и процедуры, которые позволяют компьютерам и прикладным программам обмениваться информацией.

# ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ

В настоящем отчете о НИР применяют следующие сокращения и обозначения: //TODO: правки

MVC – Model View Controller, архитектурный паттерн проектирования

ПО – программное обеспечение

IoC – Inversion of Control, инверсия контроля

DI – Dependency Injection, внедрение зависимостей

SQL – Structured Query Language, структурированный язык запросов

JPA – Java Persistence API, спецификация Java EE и Java SE, описывающая систему управления сохранением Java объектов в таблицы реляционных баз данных в удобном виде

FTP – File Transfer Protocol, протокол передачи файлов по сети

HTTP – HyperText Transfer Protocol, протокол передачи гипертекста

TCP – Transmission Control Protocol, протокол управления передачей

IP – Internet Protocol, межсетевой протокол

TCP/IP – Сетевая модель передачи данных, основанная на протоколах TCP и IP

gRPC – google Remote Procedure Call, протокол удаленного вызова функций

ОС – Операционная система

REST – Representational state transfer, архитектурный стиль взаимодействия компонентов распределённого приложения в сети

XML – Extensible Markup Language, расширяемый язык разметки

СУБД – Система управления базами данных

DTO – Data transfer object, один из шаблонов проектирования, используется для передачи данных между подсистемами приложения. Объектом этого шаблона является объект, который содержит необходимые для передачи данных поля

URL – Universal Resource Locator, универсальный указатель местоположения документа в Сети интернет

ACID – Atomicity, consistency, isolation, durability, аббревиатура четырех требований к транзакционной системе – атомарность, согласованность, изоляцию, устойчивость

API – Application Programming Interface, программный интерфейс приложения

IDE – Integrated Development Environment, интегрированная среда разработки

# ВВЕДЕНИЕ

С каждым годом сфера информационных технологий развивается все сильнее и сильнее, из-за чего все больше рутинных для человека задач выполняют программы. Автоматизация процессов повышает уровень эффективности, а цифровизация позволяет выполнять те же задачи с меньшими трудозатратами. Работа сотрудников становится быстрее и легче выполняется.

Задача разработки программного комплекса, позволяющего взаимодействовать с реальной информацией о музейных объектах, заключается в упрощении и ускорении работы человека или группы людей, который совершают паспортизацию будущих музейных экспонатов, или работают с хранилищем существующих паспортов.

Целью данной работы является создание многофункционального, удобного и эффективного программного комплекса, который позволяет пользователю свободно работать с информацией о музейных объектах, быстрее проводить цифровизацию и иметь возможность работать с хранилищем в цифровом формате. //TODO: переделать алгоритм

Для реализации цели работы выделим следующие задачи:

1. выбор требуемого стека технологий на основе проведенного анализа;
2. выбор архитектурного подходов для разработки приложения на основе проведенного анализа;
3. проектирование архитектуры и функционала приложения;
4. создание модели базы данных приложения;
5. выбор инструментов, библиотек и фреймворков для разработки;
6. разработка серверной части приложения;
   1. репозитории;
   2. сервисы;
   3. контроллеры и endpoints;
   4. mapper;
   5. компонент перехвата исключительных ситуаций;
   6. система проверки API swagger;
7. разработка стороны клиента;
   1. идея реализации шаблона MVVM;
   2. модель, представление и ViewModel;
   3. граф навигации экранов;
   4. авторизация;
   5. добавление;
   6. сканирование;
   7. модификация;
   8. списание;
   9. хранение;
   10. . инвентаризация.
8. проведение эксперимента и оценка результатов.

Объектом исследования являются архитектурные решения создания крупного программного комплекса с применением актуальных технологий и подходов в области разработки серверных сторон и мобильных приложений.

Предметом исследования является создание программного комплекса, позволяющего взаимодействовать с информацией об экспонатах при помощи и работать с цифровым хранилищем.

Актуальность исследования заключается в том, что до сих пор во многих организациях процесс проведения паспортизации является медленным и неэффективным. Человек вынужден заполнять физическую карточку, либо заполнять электронную версию (в разрозненных форматах). При поиске необходимого паспорта экспоната человек вынужден совершать поиск в архиве. Данный способ является не только долгим, но и трудоемким для работника. Поэтому стоит задача создания более эффективного и простого способа проведения паспортизации и взаимодействия с информацией о музейных объектах.

# ОСНОВНАЯ ЧАСТЬ

## 1 ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

### Обзор проблематики

Согласно статье 6 Федерального закона о музейном фонде Российской федерации и музеях в Российской федерации, музейные предметы и коллекции, независимо от того, в чьей собственности они находятся, должны проходить государственный учет. Согласно закону, государственный учет осуществляется в следующих случаях:

1. при включении музейных предметов и музейных коллекций в состав Музейного фонда Российской Федерации;
2. при исключении музейных предметов и музейных коллекций из состава Музейного фонда Российской Федерации;
3. при внесении изменений в сведения о музейном предмете или музейной коллекции из состава Музейного фонда Российской Федерации;
4. при осуществлении сделок с музейными предметами и музейными коллекциями, включенными в состав Музейного фонда Российской Федерации;
5. в других случаях, предусмотренных законодательством Российской Федерации

Государственный учет в свою очередь состоит из двух этапов: первичного государственного учета (далее – первичный учет), и централизованного государственного учета. При первичном учете проводится экспертиза культурных ценностей в целях отнесения их к музейным предметам и музейным коллекциям (далее – экспертиза культурных ценностей). При исследовании объекта экспертизы устанавливаются такие параметры, как автор (изготовитель) объекта, материал, период и регион создания, технологии исполнения и технические средства, используемые для создания объекта, а также состояние сохранности. В зависимости от особенностей будущего музейного объекта могут устанавливаться иные сведения.

Полученные данные из завершенной экспертизы заносятся в карточки музейных объектов (далее – паспорта), прикрепляются фотографии музейного объекта, позволяющие в полной мере идентифицировать объект. Также сотрудник вносит соответствующую запись в книгу поступлений необходимого фонда музея.

Во многих музеях процесс первичного учета и заполнение паспортов, проводящийся, к примеру после передачи найденных артефактов на хранение, может занять длительное время. На данный предмет принято заполнять паспорт либо в виде электронного Word-документа, либо в физическом формате, а именно бумажный экземпляр размером 10,5×14,5 см. С течением времени может образоваться емкий архив, в котором тяжело ориентироваться, или большое хранилище бумажных документов, которое необходимо содержать в идеально отсортированном порядке, для того чтобы иметь возможность поиска необходимого паспорта.

Текущие электронные версии паспортов неудобны в обращении, так как во многих музеях они просто хранятся на одном компьютере, а физические паспорта со временем могут прийти в негодность. Также возникают трудности с поиском необходимых документов, о которых говорилось ранее.

Работа с такими форматами музейных паспортов с течением времени требует все больше времени и усилий. Затраты времени и сил можно избежать, если собрать все паспорта в одно электронное хранилище. С наличием такого хранилища отпадет необходимость держать большие бумажные архивы или массивы документов на компьютере, т.к. все музейные паспорта будут храниться на сервере, а при необходимости работы с паспортами (создание новых при поступлении артефактов, редактирование информации о существующих объектах, или удаление паспортов в случае исключения музейного объекта из фонда) достаточно будет иметь мобильное устройство и соответствующее приложение.



Рисунок 1 – Пример музейного паспорта объекта

Рисунок 2 – Изображение QR-кода

Из вышеописанного следует актуальность данной технологии для хранения информации и взаимодействия с ней в рамках задачи создания единого электронного хранилища. Далее будут описаны основные технологии для создания мобильных приложений и подходы по их проектированию.

### Технологии для создания серверной части

#### Язык и фреймворк

Основой любого крупного приложения является его серверная часть. Язык программирования Java является одним из основных языков разработки серверной части. У него есть множество достоинств:

1. наличие строгой типизации и понятной семантики языка;
2. объектно-ориентированный подход;
3. высокая безопасность;
4. высокая производительность;
5. стабильность, надежность и обратная совместимость с предыдущими версиями;
6. независимость от аппаратной части и операционной системы. JVM позволяет переводить программу в байт код, который доступен любому компьютеру;
7. удобные и эффективные сетевые возможности. Имеется обширная программная библиотека для передачи данных по самым распространённым протоколам: FTP, HTTP, TCP/IP. Работает механизм вызова удалённых методов gRPC.

Java является одним из самых распространенных языков программирования, что обеспечивает огромную базу готовых решений и сайтов для поддержки, где можно найти ответы практически на любые вопросы.

Самым распространенным фреймворком для написания серверной части приложения на Java является Spring фреймворк. Он обладает множеством преимуществ:

1. может использоваться для разработки любого Java-приложения;
2. многомодульный фреймворк;
3. может применять для разработки различных слоёв приложения;
4. реализует принцип инверсии управления (IoC). Данный фреймворк хорошо известен слабой связностью своих модулей. Это помогает встраивать и извлекать модули из приложений, когда это необходимо. Поэтому, когда требуется исправить ошибки в определённом компоненте приложения, можно заменить этот модуль другим. При этом приложение продолжит работать;
5. внедрение зависимости позволяет модулям иметь слабую связность и помогает извлекать зависимости от конкретных объектов;
6. Spring фреймворк позволяет создавать и управлять конфигурациями, а также жизненными циклами объектов приложения.

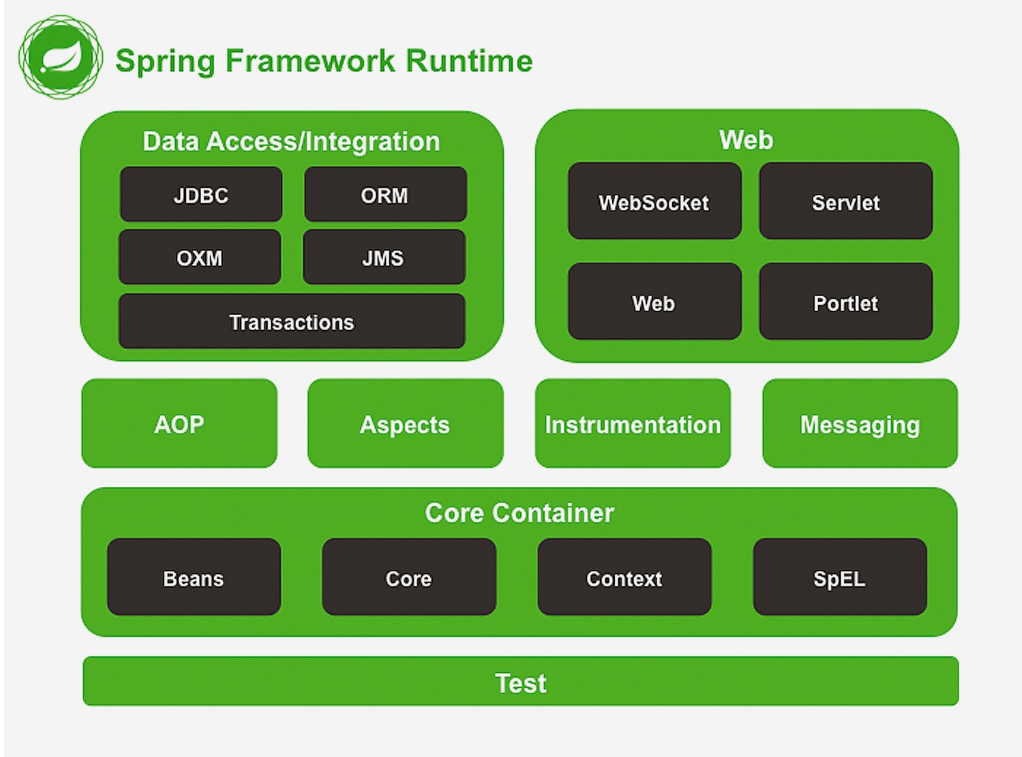


Рисунок 3 – Структура Spring framework

Spring состоит из почти 20 фреймворков таких как Spring Data, Spring AOP, Spring Boot, Spring MVC и так далее. Одной из основных особенностей Spring фреймворк является управление bean-объектами и возможность внедрения зависимостей (Dependency Injection), используя инверсию контроля (Inversion Of Control). Благодаря чему можно создавать объекты налету, без явного вызова конструктора. Такой подход облегчает процедуру создания объектов, очищает написанный код, а также гарантирует, что запрошенный объект будет создан.

Взаимодействие с SQL и NoSQL базами данных в Spring происходит благодаря Spring Data фреймворк. Он включает в себя набор интерфейсов для работы с данными через JPA Entity.

Серверную часть, написанную на Spring, зачастую используют как веб-сервис. Взаимодействие клиентов и сторонних приложений с веб-сервисами обеспечивается посредством сообщений, основанных на определенных протоколах и соглашениях. В качестве соглашения во многих проектах используется REST, в качестве протокола – HTTP/HTTPS.

Конфигурация зависимостей Spring зачастую вызывает сложности, особенно, если приложение крупное. Чтобы решить данную проблему, используется Spring Boot – дополнение к основному фреймворку, которое облегчает и ускоряет разработку. Spring Boot подключает комплекс утилит для автоматизации настройки и позволяет:

1. упаковать все зависимости в начальные пакеты (starter-пакеты), за счет чего не нужно писать однотипные скрипты для загрузки зависимостей и старта проекта. Starter-пакеты представляют собой набор удобных дескрипторов зависимостей, которые можно включить в свое приложение. Это позволит получить универсальное решение для всех, связанных со Spring технологий, избавляя от лишнего поиска примеров кода и загрузки из них требуемых дескрипторов зависимостей;
2. после выбора подходящего starter-пакета, Spring Boot попытается автоматически настроить Spring-приложение на основе добавленных jar-зависимостей;
3. если нужно использовать отдельный HTTP-сервер, для этого достаточно исключить зависимости по умолчанию. Spring Boot предоставляет отдельные starter-пакеты для разных HTTP-серверов.

#### Базы данных

База данных является основой хранилища музейных паспортов.

Информация в реляционных базах данных организованны в виде набора таблиц, состоящих из столбцов и строк. В таблицах хранится информация об объектах, представленных в базе данных. В каждом столбце таблицы хранится определенный тип данных, в каждой ячейке – значение атрибута. Каждая стока таблицы представляет собой набор связанных значений, относящихся к одному объекту или сущности. Каждая строка в таблице может быть помечена уникальным идентификатором, называемым первичным ключом, а строки из нескольких таблиц могут быть связаны с помощью внешних ключей. Данные могут быть получены из таблиц при помощи структурированного языка запросов SQL.

К наиболее распространенным представителям класса реляционных баз данных относятся:

1. MySQL;
2. PostgreSQL;
3. MariaDB;
4. Microsoft SQL Server;
5. Oracle.

В данной работе будет идти речь про бесплатную СУБД PostgreSQL. Главный серверный процесс управляет файлами баз данных, принимает подключения клиентских приложений и выполняет различные запросы клиентов к базам данных. Эта программа сервера базы данных называется postgres. У каждого серверного процесса есть локальная память, в ней лежит кэш каталога, планы запросов, рабочее пространство для выполнения запросов и так далее. Когда клиент делает запрос к базе данных, выделяется новый postgres, который обрабатывает все запросы клиента. Максимальное число пользователей в сессии по умолчанию – 100.

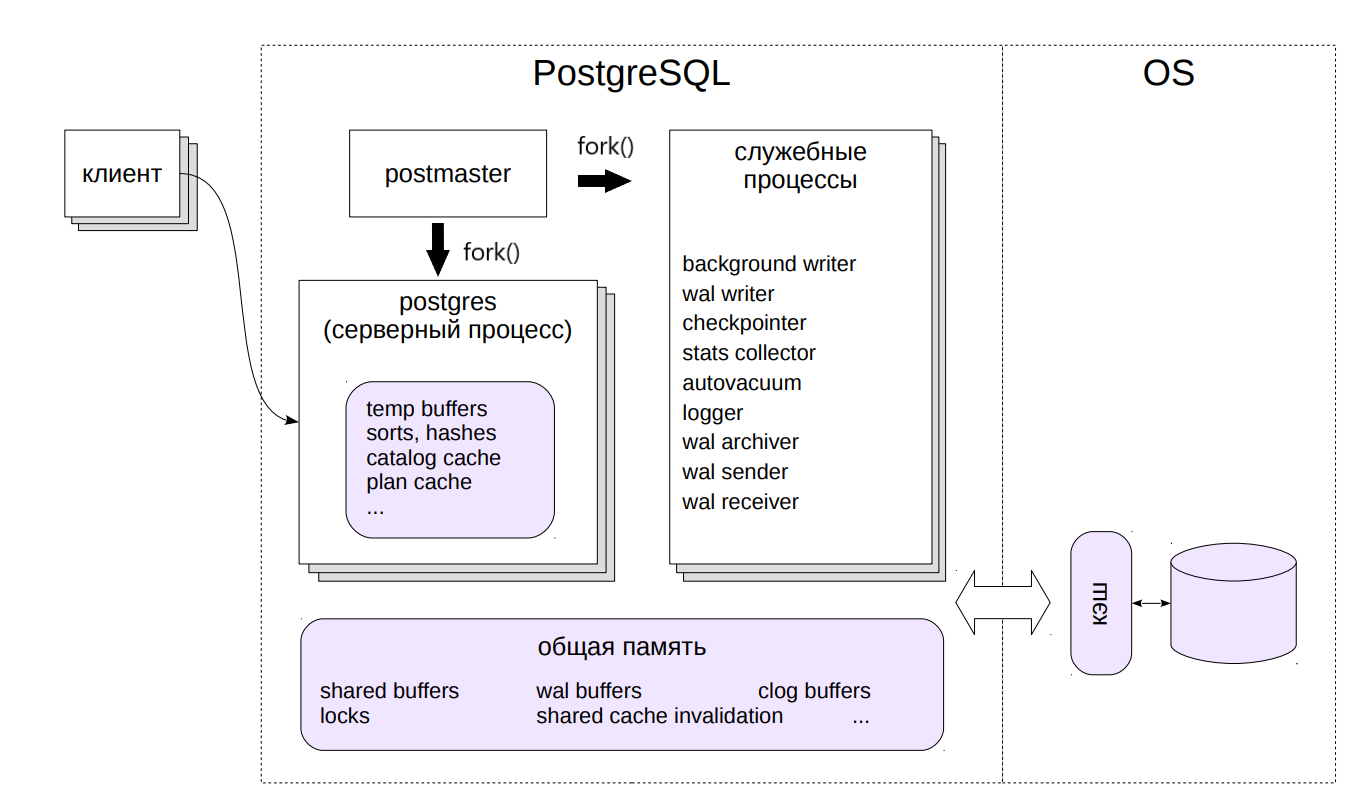


Рисунок 4 – Схема процессов и памяти PostgreSQL

Для сглаживания скорости работы памяти и дисков в PostgreSQL используется буферный кэш. Он состоит из массива буферов, который содержат блоки данных (страницы по 8 Кб).

Опишем преимущества PostgreSQL:

1. свободный доступ;
2. можно установить на любую платформу: Linux, Windows, macOS. Причем дополнительно ничего скачивать не нужно;
3. поддерживает разные форматы данных, например, сетевые адреса, данные в текстовом формате JSON, геометрические данные для координат геопозиций и многие другие;
4. размер базы данных в PostgreSQL не ограничен и зависит от того, сколько свободной памяти есть в месте хранения: на сервере, локальном компьютере или в облаке. Максимальный размер таблицы – 32 терабайта. Одна строка в базе данных не может превышать 1,6 терабайт, а максимальный размер одной ячейки – 1 гигабайт;
5. соответствует требованиям ACID (Atomicity – атомарность, Consistency – согласованность, Isolation – изолированность, Durability – устойчивость);
6. поддерживает все функции современных баз данных: оконные функции, вложенные транзакции, триггеры, хранимые процедуры, индексы, представления, материализованные представления, роли и так далее;
7. есть свой, более простой, диалект языка, который сокращает и упрощает ввод некоторых команд;
8. поддерживает логическую репликацию данных. Копия базы может находиться на другом сервере. При логической репликации любые изменения синхронизируются во всех копиях базы данных вне зависимости от места их хранения. Это значит, что везде будет храниться одна версия базы данных;
9. можно без потерь перенести данные из другой СУБД.

Благодаря всем вышеописанным преимуществам и доступности PostgreSQL является одной из самых популярных, современных и быстро развивающихся систем управлений баз данных, которая используется во многих крупных проектах.

### Технологии для создания мобильного приложения

Прежде, чем разрабатывать приложение, необходимо определиться с набором технологий и средствами разработки.



#### Операционная система для разработки

Разработка мобильного приложения на данный момент происходит под две наиболее распространенные операционные системы: Android и iOS. По статистике на 2023 год **[9]**, Android продолжает сохранять свое доминирование на мировом рынке смартфонов с долей рынка 79,2%. Доля iOS составляет 20,4%.

Популярность можно объяснить рядом факторов. Во-первых, Android является операционной системой с открытым исходным кодом. Это означает, что производители смартфонов могут адаптировать ее под свои нужды. Это позволяет использовать Android на широком спектре устройств, от бюджетных смартфонов до флагманских устройств высокого класса, что позволило ей занять большую долю рынка. Кроме того, устройства на базе Android, как правило, более доступны по цене, чем их аналоги на базе iOS. Процесс разработки на Android является более доступным, так как, в отличие от создания приложения на iOS, нет необходимости в покупке MacBook. Так как очень много разработчиков занимаются созданием приложений под Android, то гораздо больше возможностей найти ответ на возникший вопрос.

#### Подходы создания пользовательского интерфейса

В операционной системе Android существует два подхода создания интерфейса приложения: XML и Jetpack Compose. Первый подход считается традиционным, второй – современным. Чтобы сделать выбор, необходимо рассмотреть каждый из них по отдельности.

XML подход является классическим в разработке пользовательских интерфейсов приложения под Android и использовался на протяжении многих лет. Это язык разметки, который позволяет описывать структуру и внешний вид пользовательских интерфейсов в иерархическом формате. XML предлагает широкий спектр компонентов и атрибутов пользовательского интерфейса, которые можно настроить для создания макетов. Ключевые особенности:

1. XML позволяет вкладывать компоненты пользовательского интерфейса друг в друга, создавая иерархическую структуру, отражающую желаемый макет пользовательского интерфейса;
2. XML предоставляет широкий спектр компонентов и атрибутов пользовательского интерфейса, предоставляя гибкость при проектировании сложных и интерактивных пользовательских интерфейсов;
3. с помощью XML можно отделить дизайн от бизнес-логики, обеспечивая эффективное сотрудничество между дизайнерами и разработчиками;
4. макеты XML можно повторно использовать в нескольких Activity или Fragments, обеспечивая модульность кода и уменьшение дублирования.

XML был предпочтительным подходом к пользовательскому интерфейсу при разработке Android по нескольким причинам:

1. XML получил широкое распространение в экосистеме Android, что упрощает поиск ресурсов, учебных пособий и поддержки сообщества;
2. популярные интегрированные среды разработки (IDE), такие как Android Studio, предоставляют визуальные редакторы макетов XML, позволяя визуально проектировать интерфейсы и просматривать их в режиме реального времени;
3. XML имеет развитую экосистему библиотек, инструментов и платформ, которые улучшают разработку пользовательского интерфейса;
4. макеты XML совместимы со старыми версиями Android, что обеспечивает обратную совместимость.

Jetpack Compose – это современный подход к созданию пользовательских интерфейсов для приложений на Android. Jetpack Compose использует возможности языка программирования Kotlin для обеспечения декларативного и эффективного способа создания пользовательских интерфейсов. Он предлагает более интуитивный и краткий синтаксис по сравнению с XML, что позволяет создавать пользовательские интерфейсы с меньшим количеством кода и большей гибкостью. Для примера продемонстрируем, как выглядит один и тот же код, написанный разными подходами:



Рисунок 5 – Сравнение написания кода разными подходами

Как можно заметить по Рисунку 5, Jetpack Compose выглядит гораздо лаконичнее и интуитивно понятнее.

Jetpack Compose представляет несколько ключевых функций, которые расширяют возможности разработки пользовательского интерфейса:

1. с помощью Jetpack Compose можно описать, как должен выглядеть пользовательский интерфейс, исходя из его текущего состояния, а не обязательно определяют каждый компонент пользовательского интерфейса. Этот декларативный подход упрощает разработку пользовательского интерфейса и сокращает количество шаблонного кода;
2. Jetpack Compose поставляется с мощным компилятором, который оптимизирует рендеринг пользовательского интерфейса, эффективно перекомпоновывая только те компоненты, на которые влияют изменения состояния, что приводит к повышению производительности;
3. Jetpack Compose легко интегрируется с функцией предварительного просмотра в Android Studio, предоставляя визуальную обратную связь в режиме реального времени во время написания кода
4. Jetpack Compose включает в себя богатый набор компонентов Material Design, что позволяет с легкостью создавать современные и визуально согласованные интерфейсы.

Jetpack Compose предлагает несколько преимуществ по сравнению с XML для разработки пользовательского интерфейса Android:

1. Jetpack Compose упрощает разработку пользовательского интерфейса благодаря лаконичному синтаксису и мощным абстракциям. Можно создавать сложные макеты пользовательского интерфейса с помощью меньшего количества строк кода, что приводит к повышению производительности;
2. Jetpack Compose предоставляет встроенные механизмы управления состоянием, которые упрощают обработку изменений состояния пользовательского интерфейса и создание реактивных пользовательских интерфейсов;
3. Jetpack Compose предлагает плавную интеграцию с анимацией и жестами, упрощая создание интерактивного и увлекательного пользовательского опыта;
4. Jetpack Compose – это будущее разработки пользовательского интерфейса Android, поскольку Google активно продвигает и инвестирует в его развитие. Использование Jetpack Compose обеспечивает совместимость с будущими версиями Android и доступ к новейшим функциям пользовательского интерфейса;
5. при изменении состояния какого-либо поля перерисовка происходит только того компонента, который привязан к данному полю. То есть не происходит перерисовка всего пользовательского интерфейса, а только того, что нужно обновить. Данный механизм повышает производительность и улучшает пользовательский опыт от использования приложения.

### Сравнение архитектурных шаблонов для Android разработки

Архитектурные шаблоны проектирования определяют структуру приложения, задают его логику и помогают разработчику понять, как устроен продукт изнутри. Перед разработкой любого приложения необходимо выбрать наиболее подходящий архитектурный паттерн. Правильный выбор шаблона позволит создать не только качественное, но и легко масштабируемое, поддерживаемое и тестируемое приложение, над которым удобно работать команде разработчиков. Самыми распространенными архитектурными шаблонами проектирования являются:

1. MVVM
2. MVI
3. MVP
4. MVC

Как можно заметить, в каждом из вышеперечисленных паттернов есть «M» – Модель (Model) и «V» – Представление (View). Модель отвечает за данные приложения, это может быть локальная база данных, удаленный сервер, к информации которого можно получить доступ через API. Представление позволяет отображать данные, используя пользовательский интерфейс, в Android за это отвечают Fragments, Activities, Composables. Все архитектурные шаблоны имеют прослойку данных и их отображения, различия появляются с точки зрения соединения этих двух прослоек. Разберем каждый паттерн подробнее.

#### MVVM

В шаблоне проектирования MVVM есть ViewModel (часть VM в аббревиатуре MVVM), которая отвечает за бизнес-логику приложения. Класс ViewModel содержит UseCases и состояния (states) данных для отображения в слое представления. UseCase – это некоторое конкретное действие с данными, он содержит в себе интерфейс репозитория, имплементация которого взаимодействует с базой данных или удаленным сервером. Механизм работы ViewModel заключается в том, что происходит взаимодействие с слоем Model при помощи UseCase, и полученный результат отправляется в State. State же уведомляет слой представления о том, что появились изменения и нужно обновить пользовательский интерфейс. Специфика шаблона MVVM в том, что каждое состояние является независимой единицей, то есть в классе ViewModel находятся самостоятельные поля, в которые кладутся данные для отображения. Это является большим преимуществом, поскольку есть возможность комбинировать поля состояний в нужные нам классы и свободно изменять их значения.

Еще одним существенным преимуществом MVVM является сохранность данных, если приложение переходит в состояние destroy. Как известно, при смене ориентации экрана мобильного телефона происходит последовательная смена состояний приложения, как указано на Рисунке 6. И после состояния destroy все сохраненные данные в состояниях удаляются, и интерфейс собирается заново, но уже с пустыми полями.

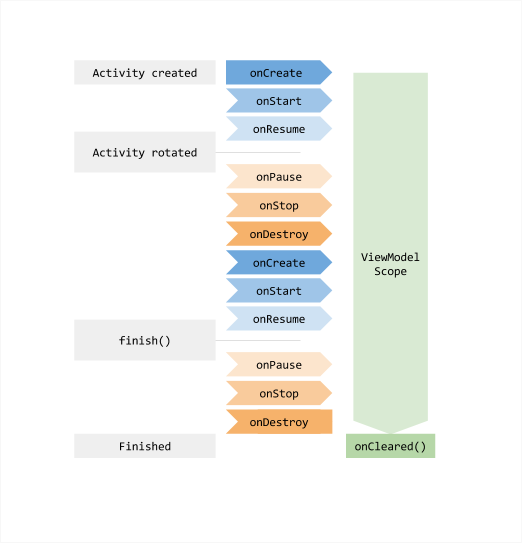


Рисунок 6 – Жизненный цикл приложения Android

В MVVM не возникает проблемы гонки данных, поскольку используется State<Flow>.

Важным положительным моментом MVVM является его хорошая совместимость с одним из преимуществ Jetpack Compose: перерисовка только конкретных компонентов, а не всего экрана. Так как в данном шаблоне архитектурного проектирования для каждого поля есть свое состояние, то при изменении значения этого состояния произойдет перерисовка только того компонента, чье отображаемое значение было изменено.

Недостатком MVVM является большое число состояний в ViewModel, в итоге размер класса увеличивается, когда необходимо добавить новое поле для пользовательского интерфейса. Из-за этого поддерживать существующий код становится сложнее.

Схема архитектурного паттерна MVVM выглядит следующим образом:



Рисунок 7 – Структура паттерна MVVM



#### MVI

MVI очень схож с MVVM. Буква «I» означает намерение (Intent).

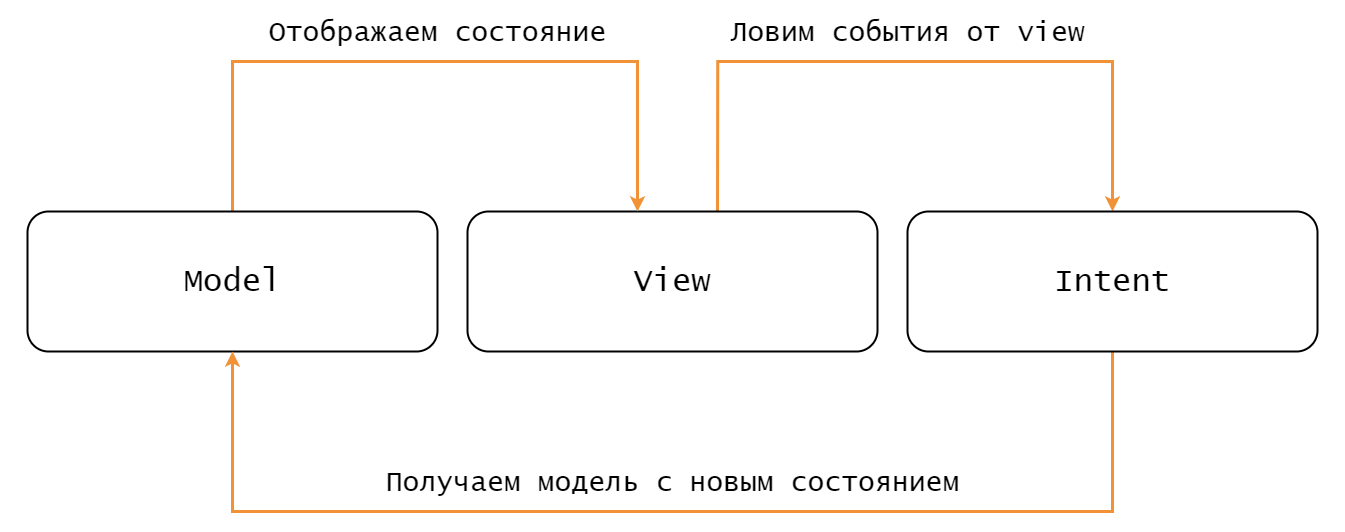


Рисунок 8 – Структура паттерна MVI

При взаимодействии с пользовательским интерфейсом, например, при нажатии клавиши происходит отправка сигнала – намерения. Класс, отвечающий за бизнес-логику, в Android это наследник от ViewModel, получает экземпляр намерения и определяет, что дальше нужно сделать. На первый взгляд данный шаблон схож с паттерном MVVM, но есть одно отличие. В MVI есть только «один источник правды» (single source of truth), то есть существует лишь одно состояние экрана, которое хранится в ViewModel. При изменении хотя бы одного поля происходит перерисовка всего экрана. В MVVM же для каждого поля есть свое состояние. Подобный подход имеет свои преимущества и недостатки.

Положительной чертой является читаемость кода, так как класс не содержит огромное множество состояний, а имеет только одно, с которым и происходят действия. Чтобы понять, за какие поля отвечает данный класс, достаточно обратиться к экземпляру состояния экрана, в котором лежат все атрибуты. В рамках Jetpack Compose в Composable функцию также не нужно передавать несколько состояний, достаточно лишь текущее состояние экрана, потому что оно уже содержит все необходимые поля.

Недостатком данного подхода является проблема гонки данных. Она объясняется тем, что если необходимо изменить состояние, то новому состоянию присваивается копия предыдущего с некоторыми измененными полями. Однако в момент присваивания планировщик Android может дать доступ к вызову другой функции «B», которая тоже должна изменить состояние, она вызовет такую же конструкцию с присваиванием и копированием, дойдет до конца. Состояние изменится. И сразу после этого вызовется первая функция «A», в которой еще не отработал оператор присваивания. Результат действия функции «B» в таком случае будет утерян, так как присвоилось состояние, которое не учитывало изменения метода «B». Эта проблема решается использованием State<Flow>.

Еще одним недостатком является передача в Composable одного большого класса с полями. С одной стороны, размер функции становится значительно меньше, что положительно сказывается на написание кода. Но в экземпляре состояния содержатся все поля, поэтому формально есть доступ к тем атрибутам, к которым данный метод не должен иметь.

Также недостатком данного подхода является проблема с сохранением состояния, если процесс был уничтожен. MVI содержит в себе состояние экрана, которое в случае аварийного завершения процесса должно быть сохранено в SavedState. Однако не во всех ситуациях необходимо подгружать прошлое состояние. Может случиться так, что на экране отображался виджет загрузки (то есть isLoading = true), и произошло аварийное завершение, состояние экрана сохранилось в SavedState. После повторного запуска приложения из SavedState будет получено сохраненное состояние. Так как в значении флага загрузки стоит «true», на экране будет крутиться виджет загрузки. Но его в действительности не должно быть, потому что процесс был аварийно завершен. Данная проблема может быть решена явным обновлением некоторых полей в значения по умолчанию, однако это лишняя работа, которую всегда нужно учитывать для каждого функционала в приложении.

Еще одним серьезным недостатком является изменение состояния экрана. Так как в MVI состояние всего одно, то при каждой его модификации будет производиться перерисовка всего пользовательского интерфейса. Данный недостаток рушит одно из преимуществ Jetpack Compose, который позволяет перерисовывать только те элементы, значения которых были изменены, а не весь экран.

#### MVP

В разработке на Android MVP используется редко, поэтому выясним отличия данного паттерна от MVVM и MVI.

В паттерне MVP помимо классических прослоек модели и представления есть Presenter. Как можно видеть из Рисунка 9, слой Presenter имеет доступ к элементам отображения данных. В шаблонах MVI и MVVM класс ViewModel не имеет доступ к прослойке представления. Activities, или fragments, или функции composable имеют доступ к ViewModel, но не наоборот.

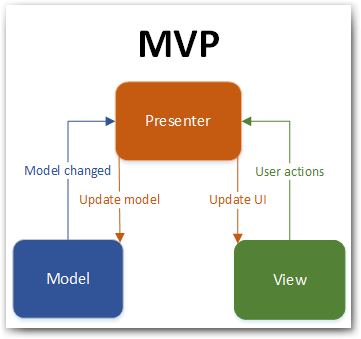


Рисунок 9 – Структура паттерна MVP

#### MVC

Аналогично шаблону MVP, MVC не часто используется в рамках создания приложений на Android. Буква «C» означает Controller.

Аналогично Presenter, контроллер не ответственен за обновление данных для слоя представления. Он лишь получает события, воздействует на данные. Слой представления же следит за изменениями в модели и реагирует на них, изменяя состояние компонентов пользовательского интерфейса.

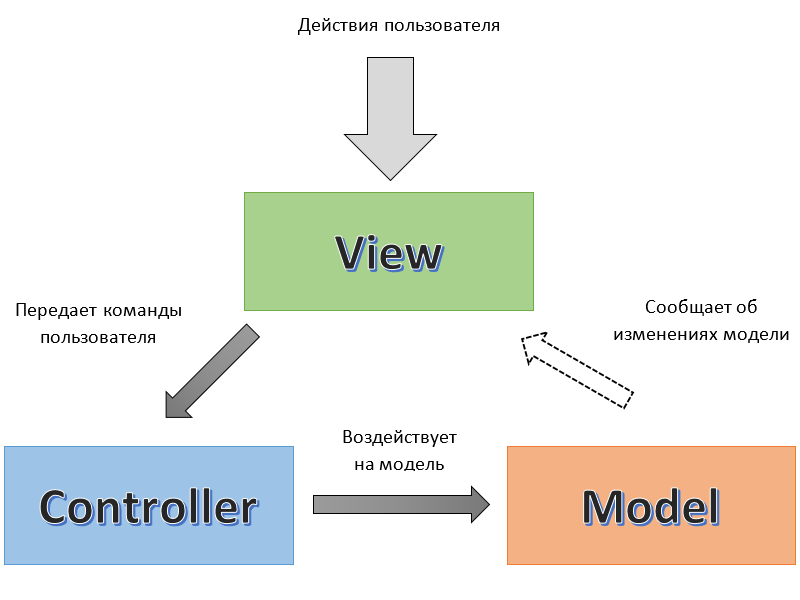


Рисунок 10 – Структура паттерна MVC



#### Вывод

По совокупности преимуществ и недостатков шаблон проектирования MVVM является во многих вопросах более предпочтительным. Проблема с аварийным завершением работы приложения в рамках данного шаблона решается гораздо проще. Также можно использовать потоковые flow механизмы, благодаря которым можно комбинировать, трансформировать, и не нужно думать о проблеме гонки данных.

## ПРАКТИЧЕСКАЯ ЧАСТЬ

Как следует из цели работы, необходимо создать программный комплекс, который позволяет пользователю свободно работать с информацией об объектах организации и быстро проводить инвентаризацию. Для достижения данной цели с практической точки зрения необходимо решить следующие задачи:

1. проектирование архитектуры и функционала приложения;
2. создание модели базы данных приложения;
3. выбор инструментов, библиотек и фреймворков для разработки;
4. разработка серверной части приложения;
   1. репозитории;
   2. сервисы;
   3. контроллеры и endpoints;
   4. mapper;
   5. компонент перехвата исключительных ситуаций;
   6. система проверки API Swagger;
5. разработка клиенткой части приложения;
   1. идея реализации шаблона MVVM;
   2. модель, представление и ViewModel;
   3. граф экранов;
   4. авторизация;
   5. добавление;
   6. сканирование;
   7. модификация;
   8. списание;
   9. хранение;
   10. . инвентаризация;
6. проведение эксперимента и оценка результатов.

### Функционал и архитектура приложения

Начальным этапом создания любого крупного приложения является проектирование необходимого функционала. Так как результатом работы является система, позволяющая взаимодействовать с информацией об объектах организации и проводить инвентаризацию, то должен следующий функционал:

1. авторизация;
2. добавление информации о новых объектах;
3. извлечение;
4. хранение;
5. модификация;
6. отображение;
7. удаление;
8. проведение инвентаризации.

Также должен быть дружественный пользовательских интерфейс мобильного приложения.

Архитектура приложения представляет из себя серверную и клиентскую части, взаимодействие между которыми осуществляется при помощи подхода RESTful API на основе протокола HTTP.

Серверная часть содержит прослойку данных, сервисов, которые работают с данными, и интерфейсов для вызова функционала – контроллеров.

В качестве архитектурного шаблона стороны клиента был выбран MVVM.

### Модель базы данных

База данных является неотъемлемой частью любого приложения. Модель базы данных должна обеспечить гибкое добавление новых сущностей, а также подходить под наибольшее количество структур организаций.

Исходя из требований, была создана следующая концептуальная и логическая модели:

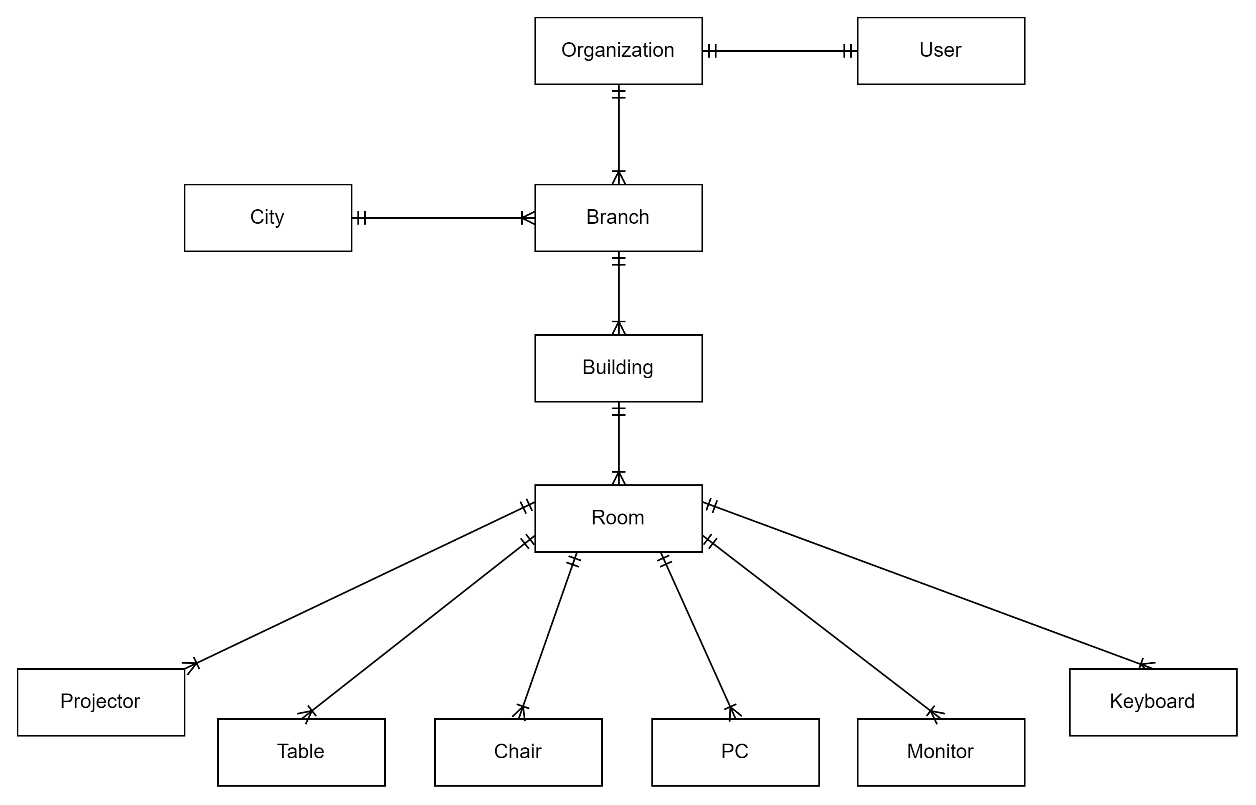


Рисунок 11 – Концептуальная модель базы данных

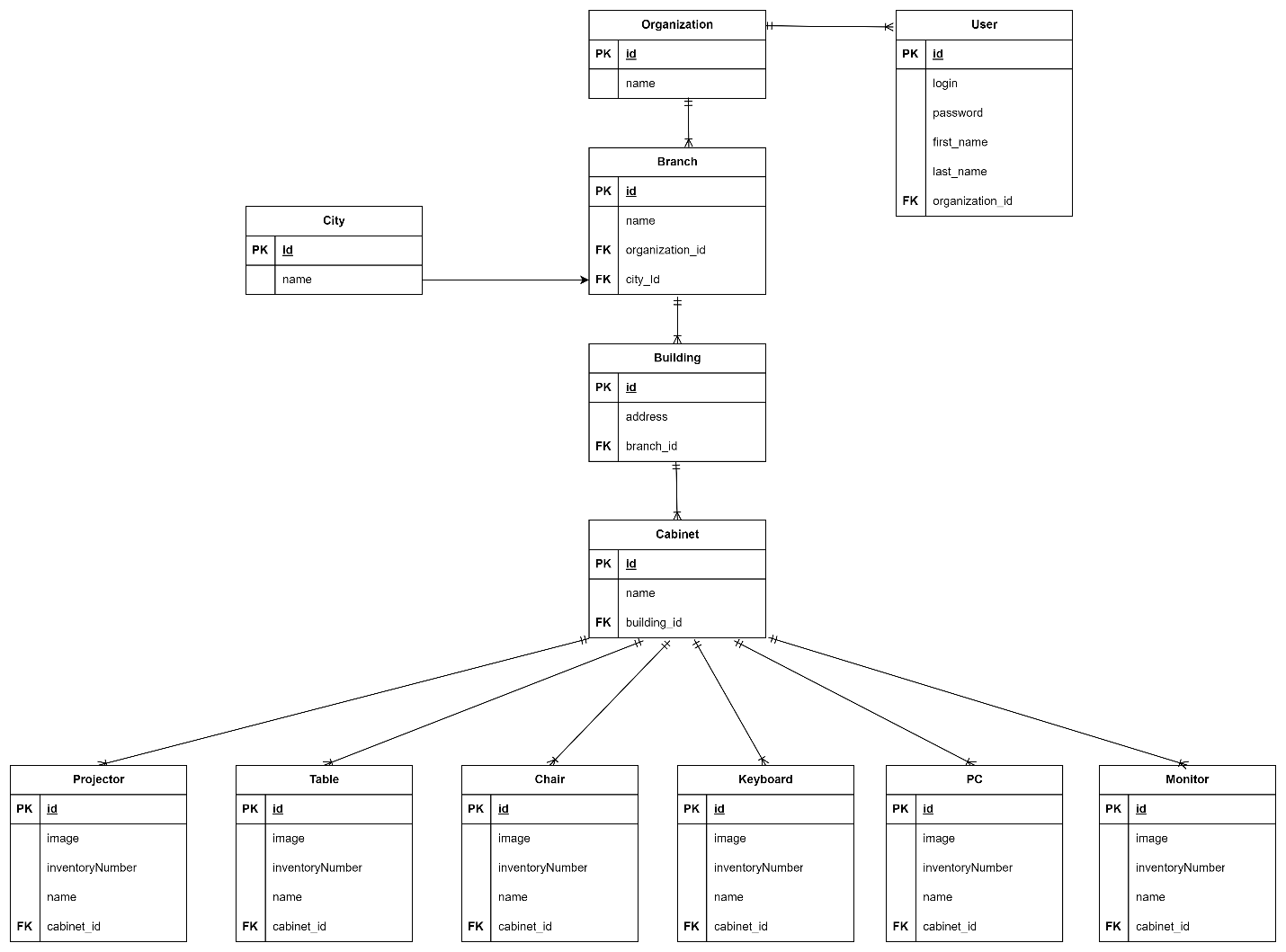


Рисунок 12 – Логическая модель базы данных

Данная архитектура позволяет использовать приложение практически для любой организации, поскольку есть возможность указать филиалы, здания, кабинеты, а также можно без трудозатрат добавлять дополнительные объекты. Любой предмет должен быть привязан к конкретному кабинету.

Также есть пользователь, который привязан к организации, в которой он должен проводить инвентаризацию. Администратор базы данных вручную добавляет пользователя, поскольку доступ к проведению инвентаризации и взаимодействию с информацией должно иметь только уполномоченное лицо.

Концептуальная модель базы данных отображает взаимосвязь между сущностями. Логическая модель раскрывает все связи и указывает первичные ключи, внешние ключи и атрибуты таблиц. У одной организации может быть множество филиалов, у одного филиала может быть множество зданий, у одного здания может быть множество кабинетов, а в одном кабинете может быть множество объектов. Также используется сущность города, которая позволят сократить количество используемой памяти. Благодаря этой сущности нет необходимости в каждый филиал явно прописывать строку города. Достаточно указать лишь уникальный идентификатор города, что уменьшает расход памяти базы.

### Выбор инструментов и технологий

В качестве СУБД была выбрана PostgreSQL, так как она обладает гибкой настройкой, широкими возможностями с точки зрения хранения больших объемов информации и безопасности SQL-запросов. Под безопасностью подразумевается строгость написания запросов и высокая надежность. Архитектура базы данных позволяет обеспечить бесперебойный доступ к информации.

Серверная часть написана на языке программирования Java, так как этот язык является быстрым, безопасным, статически типизированным, следовательно, надежным. Главным преимуществом Java является его обратная совместимость с предыдущими версиями, что гарантирует стабильность, дальнейшую поддержку приложения и обновление функционала. Данный язык программирования является одним из самых распространенных, это значит, что существует огромная кодовая база, благодаря которой можно найти решение любой проблемы.

Основным фреймворком серверной части был выбран Spring framework, так как в сочетании с Java он считается самым популярным, востребованным и современным. Благодаря Spring Boot скорость разработки приложения значительно увеличивается, поскольку нет необходимости в конфигурации и отдельном запуске сервера. Благодаря Spring Data и JPA Repository работа с базой данных становится проще, надежнее, быстрее и с меньшим количеством написанного кода, что упрощает дальнейшую разработку.

Интерфейсом взаимодействия пользователя и функционала сервера является мобильное устройство на Android. Данный выбор операционной системы был сделан из-за распространенности, доступности, большого количества современных и поддерживаемых фреймворков и библиотек.

Для создания мобильного приложения используется язык программирования Kotlin, который создан специально для разработки под Android.

Пользовательский интерфейс написан с использованием современного подхода Jetpack Compose. Он позволяет писать в функциональном стиле с меньшим количеством написанного кода, а также система перерисовки компонентов способна обновлять только измененные графические элементы, а не весь экран. Данная технология активно развивается и упрощает процесс создания компонентов интерфейса.

QR-коды создаются при помощи библиотеки Zxing zebra crossing. Данная библиотека находится в открытом доступе, поддерживает множество различных функций и является наиболее распространенной в области создания штрих-кодов.

Для внедрения зависимостей используется Dagger framework. Он позволяет автоматически внедрять зависимости в объекты, уменьшая связанность кода и делая его более гибким и легким для тестирования. Платформа предоставляет механизмы для управления временем жизни объектов, которые используются в приложении. Это позволяет эффективно использовать ресурсы и уменьшить потребление аккумулятора и памяти.

Внутреннее хранение данных в рамках телефона основывается на Room database, в основе которой лежит SQLite. Фреймворк позволяет взаимодействовать с базой данных мобильного устройства.

Взаимодействие с сервером осуществляется при помощи retrofit 2 и okhttp3. Эти библиотеки являются наиболее востребованными и популярными в рамках передачи данных между клиентом и сервером по протоколу HTTP.

Для доступа к файлу инвентаризационной описи используется библиотека Apache poi. Она предоставляет удобный интерфейс для взаимодействия с Excel таблицами.

### QR-код в приложении

Основным элементом приложения является QR-код. Идея состоит в том, чтобы на каждый объект создавать QR-код и наклеивать на предметы. Затем можно его сканировать с целью получения информации об объекте, проведения инвентаризации или изменения каких-либо полей. Все данные внутри кода хранить нельзя, потому что в таком случае он перестает быть статичным и требует регулярного перепечатывания. QR-коды используются для того, чтобы хранить ссылку на источник, и это правильный подход, так как ссылка не меняется со временем. Поэтому в системе используется следующий принцип хранения данных: каждый QR-код должен соотноситься с конкретным объектом, который хранится в базе данных, следовательно, необходимо хранить название таблицы базы данных и уникальный идентификатор предмета. Такой набор полей позволит однозначно получить предмет. А чтобы определить, какой запрос делать, необходимо использовать паттерн проектирования «Фабрика». Этот шаблон позволит по имени таблицы базы данных определить нужный endpoint. Далее выполняется запрос к серверу на известную конечную точку и извлекается нужный объект по уникальному идентификатору.

Еще одной оптимизацией для приложения является специфика проведения инвентаризации. Известно, что в таблице описи есть колонка инвентарного номера, который является статичным на протяжении всего периода жизни объекта в организации. Благодаря этим двум фактам можно принять решение поместить в QR-код еще и инвентарный номер. Эта идея позволяет значительно ускорить и облегчить процесс проведения инвентаризации, поскольку для описи важно лишь наличие объекта. Нет необходимости делать запрос на сервер и извлекать какие-либо данные, нужно лишь считать QR-код, извлечь из него инвентарный номер и сравнить его с тем, что указан в описи. Если номера совпадают, значит, это тот самый объект. Еще одним преимуществом данного подхода является отсутствие необходимости в подключении к интернету, так как не нужно делать запросы к удаленному серверу.

Однако может случиться так, что инвентарные номера двух предметов в двух разных организациях совпадают. Таким образом, возможна ситуация, когда во время описи предмет одной организации по какой-то причине находится у другой, а так как их инвентарные номера совпадают, то предмет будет отмечен как присутствующий. Чтобы решить данную проблему, необходимо в QR-код включить уникальный идентификатор организации. В таком случае сравнение будет происходить по этому полю и по инвентарному номеру. Если инвентарные номера совпадают, а id организации в QR-коде и id, к которой привязан пользователь, различаются, в таком случае предмет будет считаться отсутствующим, так как id не совпадают. Однако, здесь важно учитывать, что инвентаризацию должен проводить человек, который является сотрудником этой организации. Для хранения этой информации в QR-коде используется JSON формат. Внутри QR-кода структура будет выглядеть так: {"id":<Database\_UID>,"invNumber":<Inventory\_Number>,"orgId":<Organization\_UID>,"tableName":<Database\_table\_name>} .

### Серверная часть

Задача серверной части заключается в обеспечении доступа к данным системы. Она предоставляет интерфейс для работы с функционалом приложения и обеспечивает корректность выполняемых запросов. На сервере можно получить, добавить, изменить, удалить информацию по протоколу HTTP.

#### Репозитории

Репозитории представляют собой интерфейсы, которые могут наследоваться от JpaRepository<ModelClass, ModelUIDType> или других интерфейсов. Они предоставляют готовые SQL запросы на добавление, извлечение, обновление и удаление данных. То есть нет необходимости вручную писать запросы для каждого класса, они генерируются автоматически. Если необходимо написать какой SQL запрос, которого по умолчанию не существует, то можно добавить в интерфейс функцию, реализация которой будет автоматически создана JPA Repository на основании имени метода. Приведем пример репозитория для класса Стул:

|  |  |
| --- | --- |
| 1  2  3  4 | **@Repository**  **public** **interface** **ChairRepository** **extends** JpaRepository<Chair, Long> {  List<Chair> **findChairsByCabinetId**(Long cabinetId);  } |

Функция на строке 3 получает список всех стульев, находящихся в кабинете.

Из написанного выше следует, что репозиторий напрямую связан с сущностями базы данных. В рамках поставленной задачи инвентаризации каждый объект должен иметь инвентарный номер, название, фотографию, уникальный идентификатор и кабинет, в котором находится. Из этого следует, что лучше всего здесь использовать наследование и абстрактный класс инвентаризованного объекта, который выглядит следующим образом:

**@MappedSuperclass**

**@Data**

**@EqualsAndHashCode**

**@AllArgsConstructor**

**@NoArgsConstructor**

**public** **abstract** **class** **AbstractInventarizedEntity** {

**@Id**

**@GeneratedValue**(strategy = GenerationType.AUTO)

**private** Long id;

**@Lob**

**@JdbcTypeCode**(Types.VARBINARY)

**private** **byte**[] image;

**@NotNull**(message = "Inventory number can not be null")

**private** String inventoryNumber;

**@NotNull**(message = "Name can not be null")

**@NotBlank**

**private** String name;

**@ManyToOne**(fetch = FetchType.LAZY)

**@JoinColumn**(name = "cabinet\_id")

**private** Cabinet cabinet;

}

Наследование сокращает количество написанного кода и обеспечивает надежность того, что при создании новой сущности не нужно будет явно указывать все атрибуты заново. Класс сущности стула, будет выглядеть так:

**@Entity**

**@JsonIgnoreProperties**({"hibernateLazyInitializer", "handler"})

**@EqualsAndHashCode**(callSuper = **false**)

**@NoArgsConstructor**

**public** **class** **Chair** **extends** AbstractInventarizedEntity {

**@Builder**

**public** **Chair**(Long id, **byte**[] image, String inventoryNumber, String name, Cabinet cabinet) {

**super**(id, image, inventoryNumber, name, cabinet);

}

}

#### Сервисы

Сервисы отвечают за логику работы и являются прослойкой между данными и контроллерами. На вход в сервис приходит DTO объект, уникальный идентификатор или другая информация, необходимая для запроса. Данный класс в приватном поле имеет ссылку на репозиторий, чтобы взаимодействовать с базой данных, а также на mapper, который позволяет гибко трансформировать объект DTO класса в объект сущности базы данных. Сервисы также отлавливают ошибки в случае, если запрашиваемая информация не найдена или запрос некорректен по какой-либо причине. Пример сервиса для сущности стула представлен в **Приложении A**.

Важно учитывать, что сервисы и принимают в себя DTO объекты, и возвращают их же. То есть работа с объектом базы данных происходит исключительно в области действия сервиса. Так обеспечивается разделение ответственности между компонентами сервера. Аналогичным образом описаны все необходимые сервисы системы.

#### Контроллеры и endpoints

Контроллеры являются входной и выходной точкой в серверной части системы, доступ к нему осуществляется при помощи запроса на соответствующий endpoint. Endpoint – это конечная точка, которая позволяет обратиться извне к функционалу приложения при помощи URL адреса. Покажем структуру URL на примере контроллера для сущности стула:

/api/organizations/{orgId}/branches/{branchId}/buildings/{buildingId}/cabinets/{cabinetId}/chairs

Как можно заметить, ссылка выглядит схоже с URL в браузере. То есть путь к ресурсу строится сужением области. Для доступа к каким-либо стульям нужно сперва получить доступ к организации, затем к филиалу, затем к зданию, затем к кабинету, а только потом уже к стульям. Данный подход построения URL хорош по нескольким причинам:

1. интуитивно понятная структура;
2. возможность обеспечить безопасность доступа на каждом уровне.

Преимущество с точки зрения безопасности объясняется тем, что можно на уровне получения доступа к организации установить ограничения, потом на уровне доступа к филиалам, строениям, кабинетам. В итоге получится система с множеством уровней контроля доступа.

Если нужно получить конкретный стул, то после /chairs дописывается /{chairId}, где вместо «chairId» указывается необходимый идентификатор стула.

Класс контроллера имеет в качестве приватного поля ссылку на сервис. Также над каждым методом указывается специальная Spring аннотация, которая позволяет указать, что за HTTP запрос выполняется: GET, PUT, POST, PATCH, DELETE, HEAD или OPTIONS. На основании аннотации в пакет HTTP устанавливается тип запроса. Пример контроллера представлен в **Приложении Б**.

Важно отметить, что контроллеры работают исключительно с DTO объектами. При каждом запросе контроллер делегирует работу сервису, а сервис уже работает с базой данных и возвращает результат в виде DTO объекта обратно контроллеру, тот отправляет его на вызывающую сторону.

В качестве возвращаемого значения используется обертка над данными ResponseEntity, она позволяет упаковать результат в HTTP объект для передачи данных, чтобы можно было получить дополнительную информацию об ответе с сервера на стороне клиента.

С DTO классами используется наследование от общего потомка:

**@NoArgsConstructor**

**@AllArgsConstructor**

**@Data**

**public** **class** **AbstractInventarizedDTO** {

**protected** Long id;

**@Lob**

**protected** **byte**[] image;

**@NotNull**(message = "Name can not be null")

**@NotBlank**

**protected** String name;

**@NotNull**(message = "Inventory number can not be null")

**protected** String inventoryNumber;

**protected** Long cabinetId;

}

Тогда DTO класс конкретного объекта будет выглядеть следующим образом:

**public** **class** **ChairDTO** **extends** AbstractInventarizedDTO {}

Как можно заметить, наследование значительно ускоряет и упрощает написание кода, а также его дальнейшую поддержку.

#### Mapper

Mapper необходим для того, чтобы трансформировать объект из класса сущности базы данных в объект DTO класса. Это необходимо для того, чтобы полученный в сервисе объект отправить контроллеру, который отправит результат вызывающей стороне. DTO классы нужны для того, чтобы отправлять по сети только необходимые атрибуты. Так, например, база данных может содержать конфиденциальную информацию об организации. Очевидно, что эта информация не должна уходить в ответ запроса, нужно передать лишь те данные, которые не являются секретными, этой задачей и занимается Mapper. Подобные классы схожи с представлениями в базах данных.

Однако классы могут часто изменяться, а количество атрибутов может превышать сотни, а в некоторых проектах и тысячи. Поэтому написание трансформации вручную занимает много времени и усилий, причем из-за того, что многие классы трансформируются, Mapper всегда необходимо постоянно обновлять, а это может привести к ошибкам.

Для реализации данного функционала трансформирования объектов необходимо использовать ModelMapper и его настройку. Для этого создадим интерфейс Mapper<EntityClass, DTOClass>, который является шаблонным, потому что нам надо явно в момент компиляции указать, какие классы мы трансформируем. Интерфейс выглядит следующим образом:

**public** **interface** **Mapper**<E, D> {

E **toEntity**(D dto);

D **toDto**(E entity);

}

Далее необходимо создать абстрактный класс Mapper который имплементирует данный интерфейс, код представлен в **Приложении В**.

Здесь важно отметить функции toDtoConverter и toEntityConverter, они используют метод mapSpecificFieldsEntityToDto и mapSpecificFieldsDtoToEntity. Эти методы имеют пустую реализацию по умолчанию, так как при необходимости классы-наследники могут ее переопределить. Причем смысл этих методов заключается в том, чтобы вставить новые атрибуты в класс. В момент старта приложения для каждого Mapper класса инициализируется конфигурация, в которой явно указывается, какие атрибуты нужно пропустить, чтобы потом вручную их задать. Это бывает особенно полезно, когда в одном объекте лежит ссылка на другой объект, но нам не нужно отправлять внутренний объект, а необходимо лишь передать какие-то конкретные поля. Пример дочернего класса Mapper для класса филиала организации представлен в **Приложении Г.**

Как можно заметить, есть метод setupMapper, который конфигурирует работу конвейера трансформатора. В этом методе указывается, какие поля необходимо пропустить, они будут заданы вручную методами mapSpecificFieldsEntityToDto и mapSpecificFieldsDtoToEntity. В результате, сконфигурированный ModelMapper проведет базовую трансформацию: автоматически перенесет неуказанные в setupMapper и совпадающие по имени и типу атрибуты в новый объект класса. Поля, которые должны быть указаны вручную, будут проставлены после базовой трансформации.

#### Компонент перехвата исключительных ситуаций

Некоторые запросы могут выдавать ошибки, например, ошибка доступа или какой-либо объект в процессе поиска по идентификатору отсутствует. Подобные исключительные ситуации должны предоставлять ответ об ошибке вызывающей стороне. В Spring есть возможность делегировать перехват исключения фреймворку. Для этого необходимо создать компонент, который будет отслеживать тип исключений и обрабатывать их должным образом. Пример кода представлен в **Приложении Д.**

Каждый описанный метод позволяет перехватить соответствующее в атрибутах функции исключение и вернуть ResponseEntity в качестве ответа. ErrorInfo – это класс, который включает в себя статус код ошибки и сопроводительное сообщение, в котором указываются причина ошибки и возможные ее решения. Каждое значение статус кода показывает результат запроса:

* 1xx – информационные;
* 2xx – успешная операция;
* 3xx – перенаправление;
* 4xx – ошибка клиента;
* 5xx – ошибка сервера.

В случае возникновения ошибки нужно лишь сделать throw <тип\_исключения>, исключение будет перехвачено глобальным компонентом исключительных ситуаций.

#### Система проверки API Swagger

Для проверки работоспособности API и упрощенного создания запросов для промежуточного тестирования использовался механизм Swagger. Данная библиотека предоставляют браузерный интерфейс, в котором указываются всевозможные конечные точки с типами запросов серверной части. Достаточно лишь подключить ее и ввести в поисковую строку браузера ссылку: http://localhost:8080/swagger-ui/index.html#/. Использование Swagger значительно упрощает разработку и тестирование API.

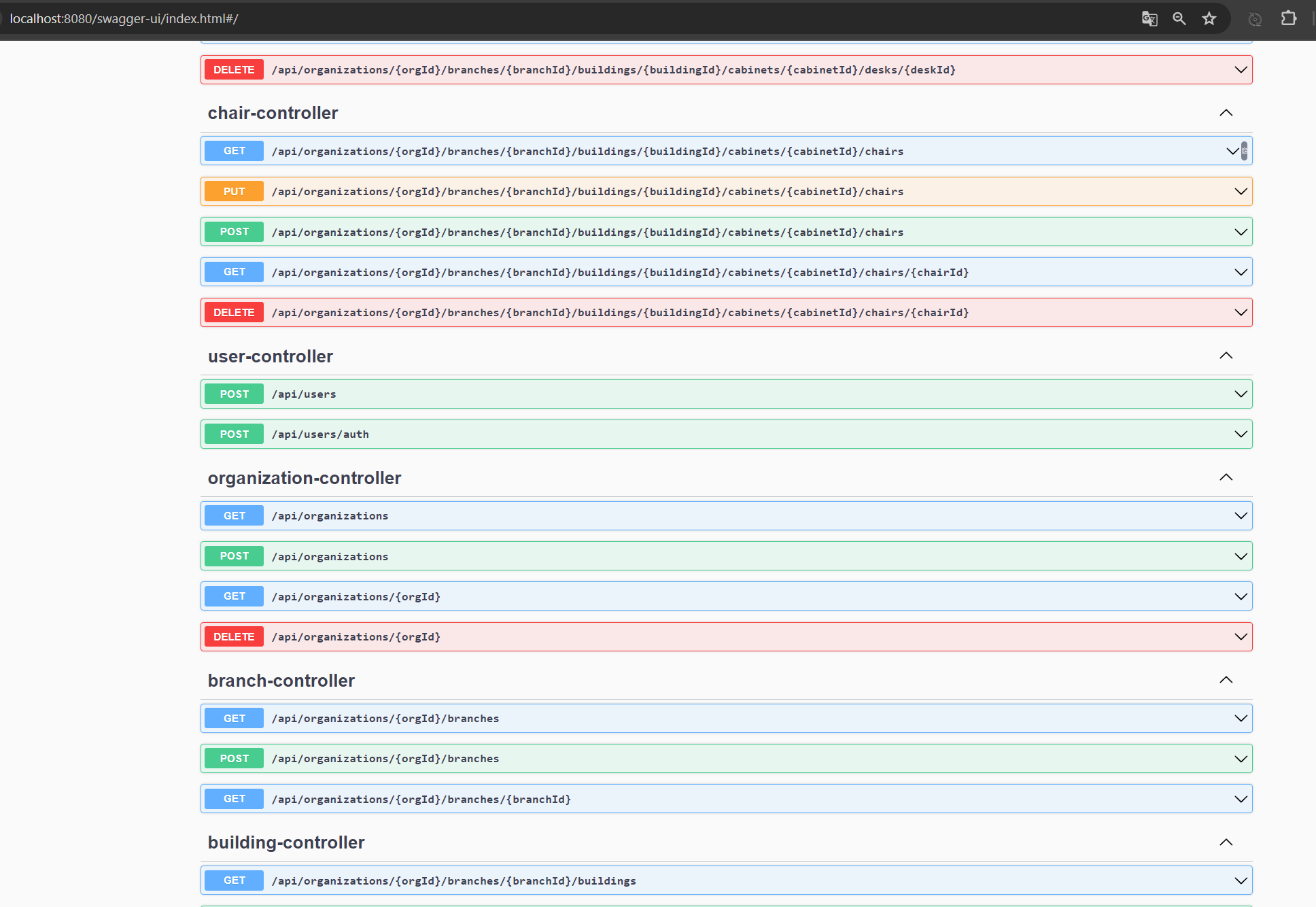


Рисунок 13 – браузерное отображение Swagger

#### Вывод

В результате работы над серверной частью были написаны API для добавления, удаления, получения и изменения следующих сущностей:

1. системный блок;
2. проектор;
3. монитор;
4. клавиатура;
5. стул;
6. парта (стол).

Также разработаны запросы для авторизации, получения информации о филиалах, зданиях и кабинетах. Для администратора базы данных есть функционал для добавления новых пользователей, организаций, филиалов, зданий и кабинетов.

### Клиентская **часть**

Клиентской частью выступает мобильное приложение. Оно должно решать поставленные задачи, а также предоставлять удобный и интуитивно понятный пользовательский интерфейс. Приложение должно обеспечивать низкое потребление заряда аккумулятора и использование оперативной памяти. Пользователь должен иметь возможность авторизоваться, добавлять объекты и создавать на них QR-коды, которые можно в последствии отсканировать и получить необходимую информацию. Также должен быть предусмотрен функционал обновления и удаления данных по QR-кодам. И самое главное – проведение инвентаризации.

#### Реализация шаблона MVVM

В основе любого крупного приложения лежит продуманная архитектура. При разработке структуры используются уже готовые и проверенные временем решения. В качестве архитектурного паттерна проектирования был выбран MVVM. Он идеально подходит для программирования под Android, поскольку для этого уже есть все необходимое. Помимо преимуществ, связанных с легкой и прочной совместимостью Android и паттерна, следует также отметить отсутствие проблем с переходами между состояниями жизненного цикла приложения. Как уже говорилось ранее, данный архитектурный шаблон обеспечивает сохранность данных внутри ViewModel при переходе в состояние Destroy, что значительно упрощает и ускоряет разработку.

Model-View-ViewModel, где в качестве Model выступает модель данных, отвечающая за работу с информацией, полученной с удаленных серверов, локальных баз данных. View отвечает за отображение данных на экране смартфона, то есть этот слой отвечает за все графические компоненты. ViewModel позволяет соединить слой модели и отображения. В ViewModel описаны состояния, которые заключают в себе данные для отображения, в этом слое и происходит вся работа с логикой приложения.

В нашем случае моделью выступает удаленный сервер с функционалом, доступ к которому происходит на основе протокола HTTP. Также используется Room фреймворк, который позволяет хранить данные на самом мобильном устройстве. Это может быть полезно для того, чтобы история сканирования не загружалась на удаленный сервер, а находилась на телефоне. Для взаимодействия с данными необходимо сперва создать классы, которые эти данные описывают. Так как клиентская часть ориентируется на серверную, то и классы сущностей должны быть соответствующими. Для того, чтобы не писать во все объекты одни и те же атрибуты, необходимо описать интерфейсы с полями и методами, которые наследуются в нужные нам классы, **смотри Приложение Е.** Важно отметить, что для определения объекта по QR-коду нужна таблица базы данных и уникальный идентификатор объекта, для предоставления подобной информации используется имплементация интерфейса QRScannableData. Этот интерфейс имеет два метода: получение имени таблицы базы данных и получение уникального идентификатора объекта. Положительной чертой интерфейсов является возможность к полиморфизму, благодаря которому с меньшим количеством написанного кода выполняется сканирование, добавление, хранение, удаление и модификация. Для получения данных необходимо также создать DTO классы с теми же полями, что в серверной части.

Представление в приложении выражено графическими компонентами, отображающими данные модели. В слое View содержится ссылка на ViewModel, таким образом, при помощи методов ViewModel и ее состояниями возможно отображение данных. Jetpack Compose предоставляет функциональный стиль написания слоя представления. Этот подход позволяет передавать callback в composable функции.

ViewModel выступает в роли связи между слоями модели и отображения. В ViewModel, как упоминалось ранее, хранятся состояния с данными модели. Эти состояния могут изменяться, при этом обновляя компоненты представления. В MVVM множество состояний, которые изменяются через State и Flow. Flow позволяет возвращать несколько значений из функции. Это может быть полезно, когда важно отображать состояния загрузки данных. В качестве примера может быть представлена следующая ситуация: необходимо получить информацию о мониторе, сканируется QR-код, делается запрос на сервер, в flow устанавливается статус «Загрузка», когда данные с сервера получены, в поток отправляется состояние «Успешно». Если произошла ошибка, то «Ошибка» с сообщением о проблеме. Каждое отправление подобной информации позволяет пользовательскому интерфейсу реагировать соответствующим образом, например, при «Загрузке» будет крутиться виджет загрузки, если «Ошибка», то всплывет диалоговое окно с информацией об ошибке. Пример представлен в **Приложении Ж.** Вызов emit позволяет отправлять в Channel информацию о состоянии, которое перехватывается ViewModel и отправляется в View. В **Приложении И** представлен пример работы с результатами перехвата flow для сканирования QR-кода. Взаимодействие между View и ViewModel происходит на основе вызовов намерений (Intent). Данный подход схож с паттерном MVI, однако в нашем случае единственного состояния экрана нет. Для того, чтобы передать намерение, необходимо создать sealed классы, которые характеризуют намерения, а затем создать подклассы намерений с уточняющими параметрами. Причем должно быть два базовых класса на каждую ViewModel: один класс для того, чтобы описать запросы от View к ViewModel, а второй – от ViewModel к View. Это сделано с целью разграничения ответственности между намерениями. Например, запросом от View к ViewModel может быть нажатие кнопки, которое должно быть обработано слоем логики. От ViewModel к View может быть требование показать окно с ошибкой из-за отсутствия запрашиваемого объекта. Код этих двух классов описан в **Приложении К.**

#### Граф навигации экранов

Граф навигации экранов позволяет концептуально отобразить структуру приложения. Он бывает полезен в момент разработки для понимания, какие уже созданные экраны существуют и можно ли их повторно использовать в разных ветвях. Граф разработанного приложения выглядит следующим образом:

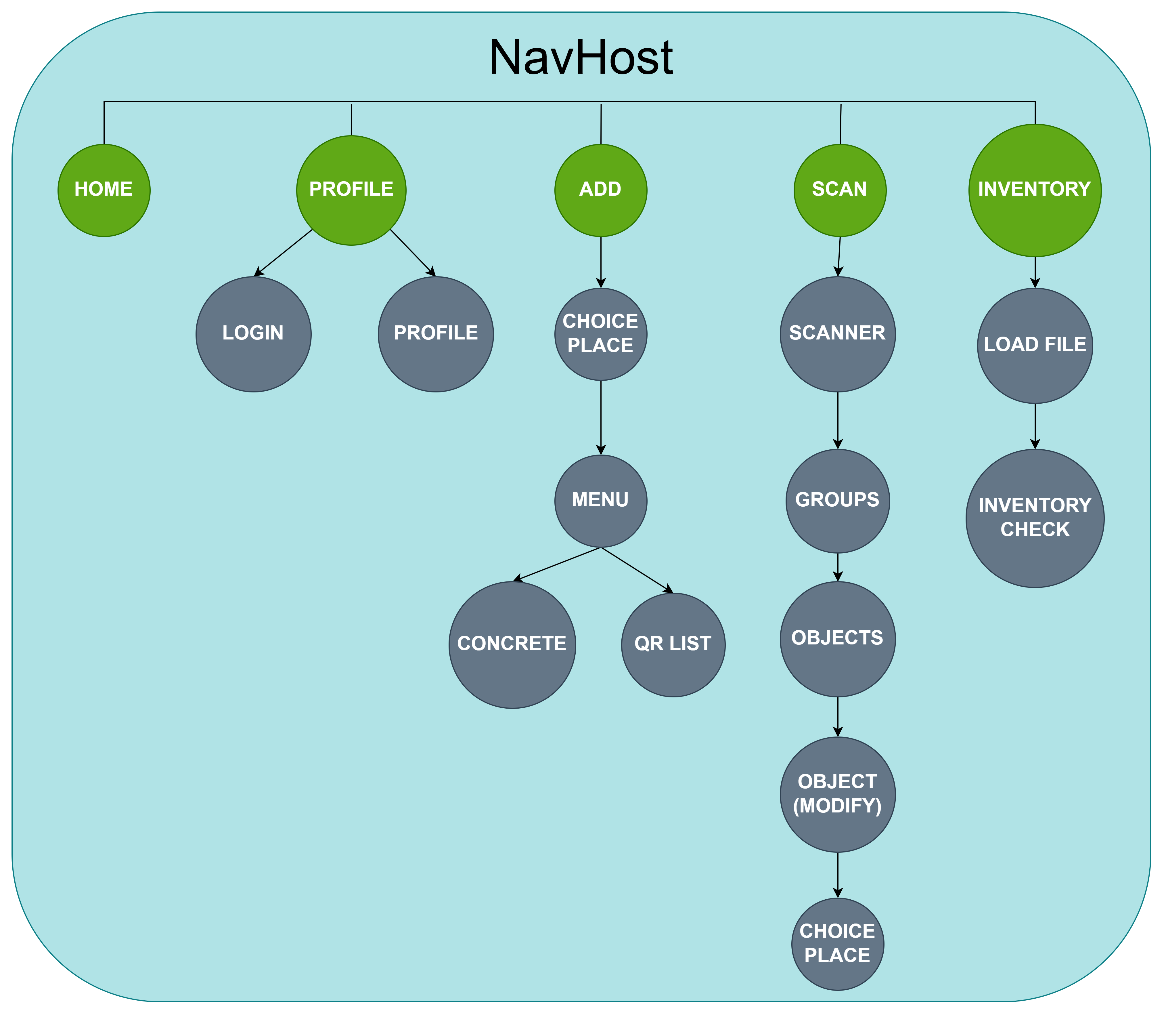


Рисунок 14 – Граф навигации экранов

NavHost представляет из себя контейнер навигации. Благодаря NavHostController возможно перемещение между узлами и ветвями графа. Причем доступ вершин, отмеченных зеленым цветом на Рисунке 14, возможен из любого места. Это достигается нижней панелью на экране мобильного телефона, которая отовсюду доступна.

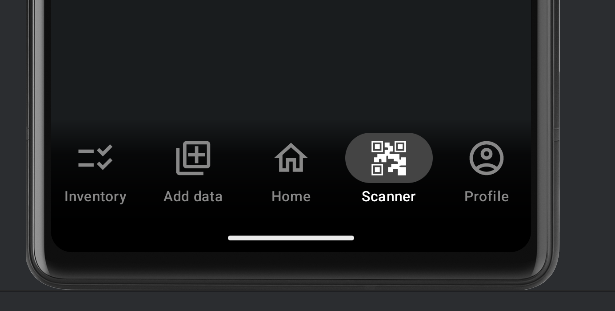


Рисунок 15 – Панель навигации между ветвями графа

Каждый серый узел на Рисунке 14 является полноценной архитектурной единицей в рамках паттерна MVVM. Практически для любого экрана есть слой модели, отображения и ViewModel. Причем важным моментом здесь является то, что каждый узел в графе изначально имеет свою область видимости. Это означает, что возникает проблема переноса данных из одного узла графа в другой, так как ViewModel при переходе уничтожается. Чтобы решить эту проблему, используется граф навигации и возможность иметь один и тот же экземпляр ViewModel для различных узлов. С программной точки зрения структура графа показана в **Приложении Л**.

Действительно, проблема сохранности ViewModel и переноса данных между экранами является ключевой в рамках создания приложения на Android. Чтобы решить эту проблему, нужно в фабрике ViewModel явно указать, из какого узла графа навигации берется экземпляр. Классический пример кода для подобной ситуации представлен в **Приложении М**. Там же приведен пример, как передавать в конструктор необходимые аргументы, которые не поставляются механизмами внедрения зависимости.

#### Внедрение зависимостей

Внедрение зависимостей необходимо для того, чтобы упростить создание экземпляра класса. Бывают ситуации, когда для одного объекта нужно создать сотни других, которые по цепочке помещаются в конструкторы, а затем в запрашиваемый объект. Из-за этого приходится писать очень много кода, причем зачастую данные внутри этих объектов не меняются, можно было бы создать один экземпляр данного класса на всю программу и использовать его. Однако без механизма внедрения зависимостей это невозможно. Так, для ViewModel используются UseCase классы, задача которых выполнить некоторое конкретное действие, например, получить стул по уникальному идентификатору. Для такого класса в конструктор должна быть передана реализация репозитория, взаимодействующего с базой данных или удаленным сервером. А репозиторий должен включать в себя интерфейс API. Отсюда следует цепочка вызовов конструкторов, инициализация которых может привлечь к неоправданному разрастанию кода. Чтобы решить данную проблему, используется фреймворк Dagger Hilt. Достаточно лишь указать конфигурацию, в которой указываются зависимости между объектами. Данный фреймворк сам создаст необходимые объекты и поместит их в нужное место. **В Приложении Н** представлен пример того, как внедряются зависимости.

В следующих главах будут описаны идеи и реализации основных функций приложения.

#### Авторизация

Чтобы использовать возможности приложения, необходимо пройти авторизацию. Для этого нужно ввести логин и пароль, которые предоставляются администратором организации.

При успешной авторизации будет доступен весь функционал приложения. В противном случае приложение будет запрашивать пройти этап идентификации. Внешний вид экранов показан в **Приложении П**.

Авторизованный пользователь является представителем конкретной организации, поэтому ему доступна информация только своей организации. Однако, как упоминалось ранее, проблема получения одного и того же экземпляра ViewModel решается путем переопределения фабрики класса, в которой указывается родительский узел графа, из которой нужно достать объект ViewModel. Но данная схема работает в рамках одной ветви графа (среди тех узлов, что находятся под зеленым узлом на Рисунке 14). Но информация об авторизованном пользователе должна быть доступна для всего функционала приложения, а запрашивать каждый раз логин и пароль пользователя при переходе в новую ветвь нельзя. Чтобы решить поставленную проблему, необходимо создать ViewModel на уровне NavHost, чтобы можно было из любого места в графе навигации получить состояние пользователя. А далее это состояние просто передается по ветвям графа с актуальным пользователем.

#### Добавление

Для взаимодействия с информацией об объектах организации необходимо их сперва добавить в систему. Чтобы это сделать, достаточно перейти на экран добавления предметов, определить кабинет, выбрать необходимый класс объекта и заполнить поля, после чего нажать кнопку «Add». В результате объект будет добавлен в базу данных сервера. На добавленный объект будет создан QR-код, который будет отображен в списке. После того, как все необходимые объекты для кабинета созданы, можно перейти на экран списка QR-кодов, выбрать для них размеры и создать файл в формате pdf. Далее этот файл можно будет распечатать, по штриховым линиям разрезать коды и прикрепить их на предмет. Причем у человека не возникнет вопроса, куда приклеивать QR-код, так как в файле будет указана вся необходимая об объекте информация: имя, инвентарный номер, класс, к которому принадлежит объект.

Далее будет подробно описан каждый шаг добавления объекта.

Сперва нужно выбрать кабинет, в котором предмет будет располагаться. Для этого нужно ввести филиал организации и здание, а только потом выбрать кабинет. Причем все поля появляются в выпадающем списке, поэтому проблемы нарушения целостности данных не будет. Это связано с тем, что на каждый выбор пользователя филиала делается запрос на сервер, чтобы предоставить список зданий, ассоциирующихся с этим филиалом. Таким же образом устроена логика извлечения списка кабинетов по отношению к выбранному зданию. Причем есть возможность смены кабинета, для этого необходимо нажать соответствующую кнопку на экране меню классов или внутри формы добавления объекта.

Внутри программы изначально задается тот набор объектов, которые подлежат обработке в рамках системы. Сгруппированные по первой букве классы этих объектов отображаются на экране меню, после чего можно нажать на конкретный класс и внести данные о предмете. После выбора конкретного класса, он сохраняется в состоянии ViewModel, отвечающей за меню классов. После чего выполняется переход на экран конкретного для добавления объекта. Выбор класса ViewModel и функции отображения основывается на значении состояния выбранного класса в меню. У каждого предмета есть своя ViewModel, это объясняется тем, что не все объекты имеют одни и те же поля. Однако каждый объект точно имеет наименование, фотографию, инвентарный номер и идентификатор кабинета. Всю работу с этой информацией и созданием QR-кода можно вынести в абстрактный класс, а дочерние классы ViewModel будут работать только со специфичными для данного объекта полями. Таким образом, обеспечивается легкая поддержка написанного кода и повышенная скорость добавления новых предметов. Код абстрактного класса ViewModel, а также одного из дочерних представлены в **Приложении Р**.

Поля наименования, инвентарного номера, кабинета и фото являются обязательными для любого предмета. Фотографию на объект можно сделать прямо из приложения при помощи камеры либо выбрать из галереи смартфона. Сделано это с той целью, чтобы обеспечить возможность сперва сфотографировать необходимые для добавления предметы, а затем вносить о них данные в удобном месте и в комфортное время.

Основой приложения является создание QR-кода. Как говорилось в разделе 1.5.4, внутрь помещается имя таблицы базы данных, уникальный идентификатор предмета в базе – эта информация уже обеспечивает однозначность определения объекта. Также есть еще два поля: инвентарный номер и идентификатор организации, причина их добавления пояснена в разделе 1.5.4. После нажатия кнопки добавления создается QR-код, который добавляется в список. Внутри списка можно изменить размер этикетки, существует 3 размера: маленький, средний, большой. Решение сделать различные размеры обосновывается тем, что объекты могут быть разными по величине. И необходимо обеспечить возможность приклеивания QR-кода на поверхность малой, средней и большой площади. После того, как размер двумерного штрих-кода выбран, можно нажать на кнопку создания pdf файла с QR-кодами для печати. Для этого необходимо выбрать каталог расположения будущего файла в системе Android, после чего ввести название документа. Файл имеет удобную структуру, в нем все коды разделены линиями, а также есть вся необходимая информация для определения объекта:

1. название;
2. инвентарный номер;
3. класс предмета.

Смысл добавления названия и класса заключается в том, что человек не может глазами считывать QR-код, поэтому нужна дополнительная информация, чтобы понять, к какому объекту он относится. Инвентарный номер добавлен в случае повреждения этикетки. Тогда инвентаризацию можно будет провести классическим способом, сверяя номера вручную.

Фотографии приложения с работой функции добавления, а также пример документа показаны в **Приложении С**.

#### Сканирование

Так как приложение работает с QR-кодами, то необходим функционал сканирования, при котором происходит получение информации об объекте. Чтобы создать камеру, способную извлекать информацию из QR-кода, используется дополнительный класс-анализатор ImageAnalysis, который встраивается в AndroidView через фабрику. Эта надстройка позволяет добавить функцию анализа на каждый кадр работы камеры. По полученному в метод изображению происходит поиск QR-кода и попытка его декодирования. Для этого используется библиотека Zxing zebra crossing. Код AndroidView и реализация анализатора представлены в **Приложении Т**. Однако в процессе реализации возникла проблема постоянного повторного анализа одного и того же QR-кода. Причина проблемы в том, что камера генерирует множество кадров, а обработчик работает на каждый кадр, поэтому выполнялось множество запросов на сервер для одного и того же штрих кода. Человек не может физически показать QR-код только на один кадр камеры. Чтобы решить данную проблему, был введен таймер дупликации, который не дает делать запросы на сервер, если QR-код на кадре совпадает с предыдущим. Таким образом, была значительно увеличена скорость работы приложения, снижена нагрузка на сервер и потребление энергии мобильного устройства.

Когда анализатор считал информацию из QR-кода и получил название таблицы базы данных и id предмета, то через паттерн проектирования «Фабрика» можно получить нужный UseCase получения объекта с удаленного сервера и сделать запрос. Для отображения данных необходимо обновить значение состояния текущего отсканированного объекта в ViewModel. Jetpack Compose обнаружит изменения и перерисует графические компоненты.

В этом случае достаточно иметь лишь одну ViewModel для всех объектов. В отличие от функции добавления, где у нас могут поля отличаться, в этом случае первичные данные отображаются у всех объектов одинаковые: наименование, инвентарный номер, фотография, филиал, здание и номер кабинета. Если нужна полная информация об объекте или нужно поменять какие-либо значения атрибутов, то необходимо добавить отсканированный предмет в специальный список. Для этого нужно нажать на обведенную красным контуром кнопку на Рисунке 16:

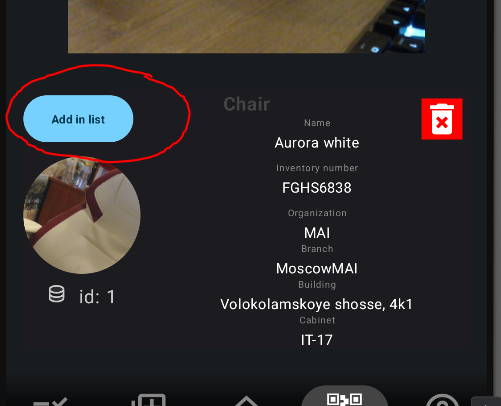


Рисунок 16 – Кнопка для добавления в список

После добавления на иконке сверху будет расти счетчик, показывающий количество объектов в новом списке:

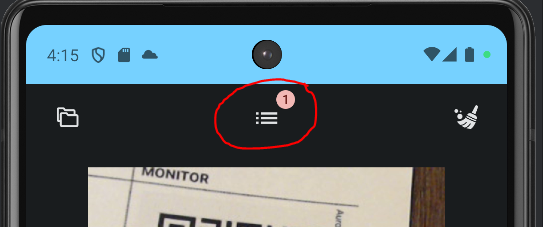


Рисунок 17 – Счетчик объектов в списке

Если все необходимые предметы добавлены, то необходимо нажать на обведенную красным контуром кнопку на Рисунке 17. Приложение запросит имя для каталога. После ввода названия будет показано окно со всеми созданными папками с отсканированными объектами. Чтобы посмотреть или модифицировать информацию о предмете, достаточно войти в один из каталогов и нажать на нужный объект.

Фотографии с демонстрацией сканирования показаны в **Приложении У**.

#### Модификация

Чтобы посмотреть или модифицировать информацию о предмете, достаточно войти в один из каталогов и нажать на нужный объект. В результате будет открыто окно, похожее на добавление. Отличие будет лишь в том, что все поля уже заполнены информацией из базы данных. Если какие-то значения атрибутов нужно поменять, то это можно сделать, после чего внизу формы нужно обязательно подтвердить изменения. Поменять инвентарный номер запрещено, поэтому допуск к вводу символов в это поле не предоставляется. В качестве реализации функции модифицирования данных использовалась существующая механика добавления. Операции по своей логике крайне схожи, отличие заключается лишь в том, что в добавлении поля для заполнения пустые, тогда как в обновлении они заполнены. Также вместо POST запроса на сервер нужно сделать PATCH. Мысль о сходстве реализаций позволяет значительно сократить объем кода. Необходимо лишь в конструктор ViewModel создания предметов добавить UseCases на обнновление. А также внутри ViewModel проверять, что за операция сейчас требуется, чтобы применить нужный UseCase. Фотографии с демонстрацией операции модификации представлены в **Приложении Ф**.

#### Списание

Чтобы списать объект из организации, достаточно отсканировать его и удалить, нажав на красную иконку с корзиной. Пример показан в **Приложении Х**.

#### Хранение

Как уже говорилось в разделе 1.5.6.6, приложение позволяет сохранять отсканированные объекты в специальных каталогах, созданных пользователем. Это сделано с той целью, чтобы не нужно было каждый раз сканировать одно и то же. Достаточно один раз отсканировать все необходимые объекты, создать именной каталог, чтобы потом работать с предметами в нем, когда и где это требуется. Эта функция может быть полезна, если окружение для сканирования является неблагоприятным для человека, но тем не менее модифицировать или проверить данные нужно. В таком случае пользователь сканирует все предметы, а изменяет их атрибуты в другом месте. Каталоги позволяют группировать объекты по известным человеку критериям. Внутри самих каталогов объекты уже сгруппированы по классам: стулья, столы, мониторы и так далее. Если нужно очистить текущий список и начать новый, то для этого на экране сканирования необходимо нажать на кисточку в правом верхнем углу экрана. Если требуется обратиться к каталогам, то нажать на иконку в левом верхнем углу экрана.

Еще одна особенность добавления в список заключается в том, что система предупреждает пользователя о дубликатах. Если человек сканирует очень много объектов, то может произойти ситуация, когда предмет по ошибке сканируется дважды или более раз. В таком случае программа предупреждает пользователя о дупликации данных:

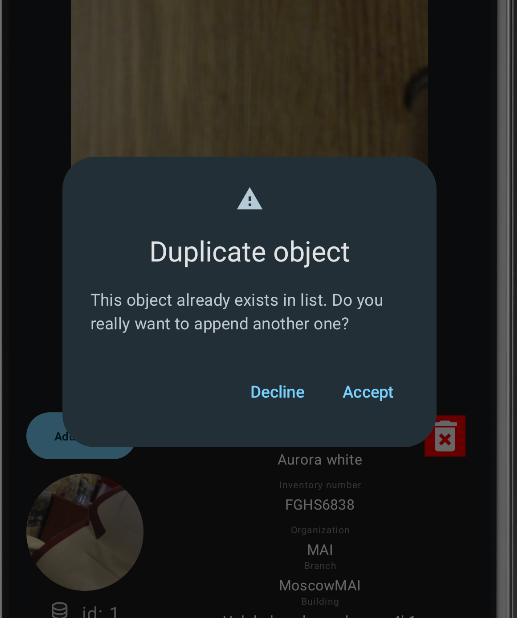


Рисунок 18 – Предупреждение о дубликате

Вся информация, связанная с каталогами, хранится внутри телефона при помощи фреймворка Room. Он предоставляет возможность взаимодействия с внутренней базой данных SQLite. Концептуальная и логическая схемы встроенной базы данных выглядят следующим образом:

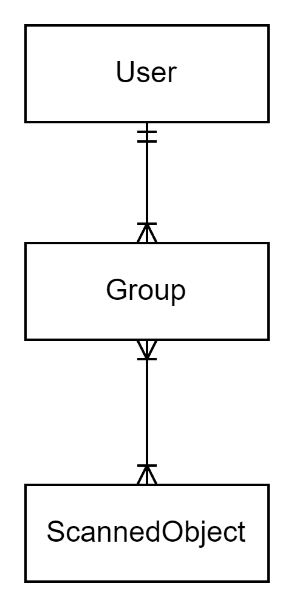


Рисунок 19 – Концептуальная модель

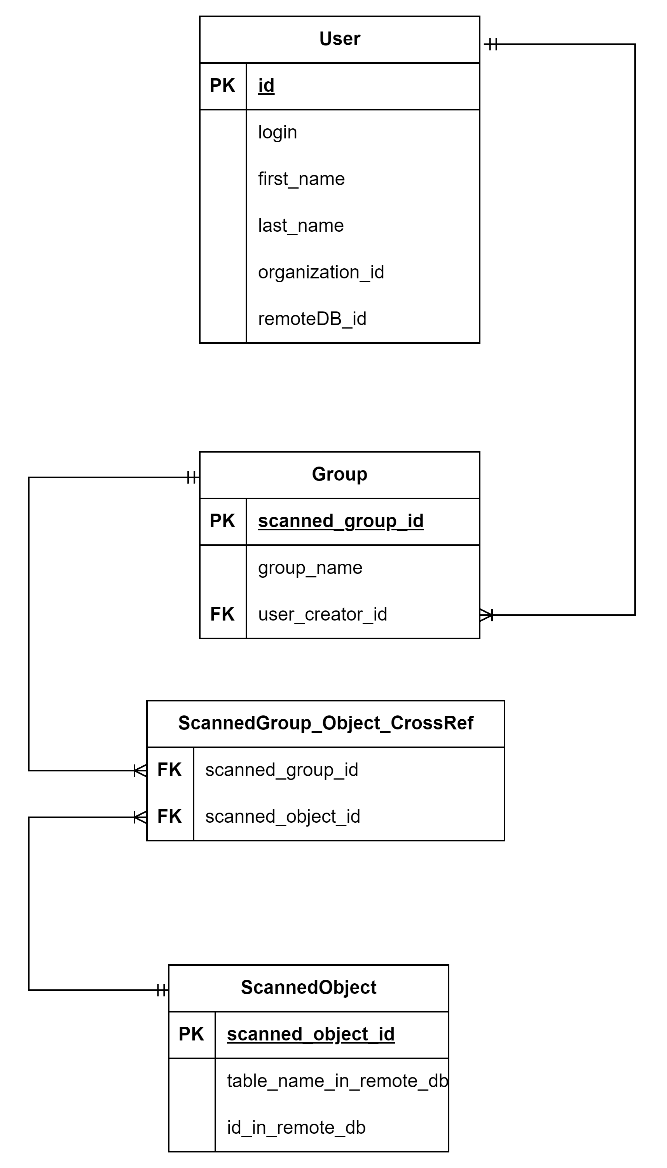


Рисунок 20 – Логическая модель

Как можно заметить, есть сущность пользователя, у которой есть множество групп, а у групп есть множество объектов. Причем один и тот же объект может находиться в разных группах и при удалении его из одной папки, он не должен исчезнуть из другой. Сущность пользователя косвенно связана с сущностью пользователя на удаленном сервере: если пользователь авторизуется, но в базе данных смартфона, данного пользователя еще нет, тогда он добавляется в базу данных. Если же пользователь уже хоть раз авторизовывался в системе, в этом случае в базе данных лишь обновляется информация.

Из схемы логической модели на Рисунке 20 следует, что в базе данных не хранится никакая значимая информация об объекте за исключением названия таблицы и уникального идентификатора, что гарантирует безопасность данных при неавторизованном доступе. Даже если будет проведена успешная атака на внутреннюю базу данных смартфона, никакой значимой информации злоумышленник не извлечет.

Есть возможность удалять каталоги и объекты из каталогов. Чтобы удалить папку, достаточно нажать на нее и удерживать палец, после чего в углу иконки появится корзина, на которую нужно нажать. После чего появится уточняющее намерения окно, в котором нужно подтвердить желание удалить каталог.

Фотографии работы с каталогами показаны в **Приложении Ц**.

#### Инвентаризация

Важнейшей функцией данной системы является проведение инвентаризации по форме ИНВ-1. Чтобы провести инвентаризацию, нужен файл описи в формате Excel. Внутри файла располагается оглавление, затем идет таблица с объектами описи и прочее. Внешний вид файла приведен в **Приложении Ш**. Проведение инвентаризации можно разделить на три этапа:

1. извлечение данных из Excel таблицы;
2. проведение описи сканированием QR-кодов;
3. занесение результатов инвентаризации в таблицу.

Чтобы извлечь данные из Excel используется библиотека Apache Poi, которая позволяет получать доступ к ячейкам. Однако местонахождение таблицы может отличаться, поскольку данный файл генерируется другой программой, и расположение колонок и строк таблиц может отличаться. Поэтому необходимо применять эвристический алгоритм поиска таблицы и позиции колонок в файле Excel. По виду файлов из **Приложения Ш** видно, что таблица начинается всегда с фразы «номер по порядку», причем в этой записи могут быть различные невидимые символы и лишние пробелы, но их можно убрать. Чтобы найти эту фразу, заметим, что таблица с объектами располагается по левому краю Excel файла. Тогда для поиска нужно итерировать по ячейкам левого края до тех пор, пока не найдем фразу «номер по порядку». Причем в случае неудачного поиска, если вдруг фраза не находится в колонке «A», то можно проделать то же самое с еще несколькими колонками. Если с учетом прохода по дополнительным колонкам таблица не была найдена, то файл считается некорректным.

Когда левый верхний угол таблицы найден, в таком случае достаточно заметить, что под каждым названием колонки есть порядковый номер от 1 до 13. В форме ИНВ-1 фиксировано количество колонок, а также каждому порядковому номеру колонки однозначно соответствует ее предназначение. Как только фраза «номер по порядку» найдена, достаточно спуститься вниз по строчкам, пока не найдем «1». Известно, что все цифры находятся в одной строке, поэтому достаточно найти лишь 13 позиций колонок. Также известно, что строки для описания объектов отстают друг от друга вниз на 1. Исходя из всего вышеописанного, можно написать алгоритм, который получает все необходимые данные из таблицы описи. Алгоритм представлен в **Приложении Щ.**

Когда все данные получены и занесены в классы программы, то можно провести инвентаризацию. Чтобы это сделать, нужно отсканировать QR-коды предметов в кабинете. Так как внутри кода находится инвентарный номер, то запросы к базе данных делать не нужно, необходимо лишь сравнить номера и идентификатор организации. Если они совпадают, тогда объект отмечается как присутствующий.

Иногда бывает, что инвентарные номера не указываются. Так случается, когда предметов больше 1, поэтому в описи не пишутся инвентарные идентификаторы. Тогда чтобы узнать, есть ли объект в помещении, необходимо отсканировать QR-код, сделать запрос на сервер, узнать наименование предмета и сравнить его с тем именем, что указано в документе описи. При совпадении предмет считается присутствующим. Пример отображения процесса проведения инвентаризации показан в **Приложении Э.**

После проведения сверки необходимо нажать кнопку завершения инвентаризации в правом верхнем углу экрана. После чего программа сама проставит фактическое наличие и фактическую стоимость в таблицу Excel. Далее человеку необходимо лишь проверить результаты описи и заполнить оставшиеся поля вручную. Результирующая таблица после инвентаризации показана в **Приложении Ю**.

## 3 РЕЗУЛЬТАТЫ РАБОТЫ



### Сферы применения программного комплекса

Разработанный программный комплекс для хранения, внедрения, извлечения, изменения информации об объектах организации и проведении инвентаризации при помощи QR-кодов может быть использован в различных контекстах. Данная разработка может использоваться для хранения и быстрого извлечения данных. Она значительно упрощает и ускоряет процесс взаимодействия человека с системой. Также данная разработка может использоваться для проведения инвентаризации. Использование QR-кодов в процессе описи требует меньше усилий, а скорость выполнения становится гораздо выше.

### Результаты разработки

В программном комплексе была реализована возможность авторизации, позволяющая получить доступ к функционалу приложения. Есть возможность добавления данных на сервер с последующей генерацией QR-кода, который можно распечатать и прикрепить на объект. При наличии QR-кода можно проводить сканирование объектов с целью получения информации, также можно изменять значения атрибутов и делать списание предмета. В мобильном приложении разработана система каталогов, которая позволяет безопасно сохранять сканированные объекты для дальнейшей манипуляции с ними. Также есть возможность проведения инвентаризации с использованием QR-кодов без необходимости использования интернета, если в описи заданы инвентарные номера проверяемых объектов.

Исходные коды серверной и клиентской частей находятся в **Приложении Я**.

### Проведение эксперимента и результаты

Для проверки эффективность созданного комплекса был поставлен эксперимент инвентаризации. Было создано 100 QR-кодов, каждый из которых был прикреплен к конкретному объекту в комнате. После чего проводилась инвентаризация тремя способами:

1. с использованием системы;
2. вручную последовательно по списку;
3. вручную по местоположению и поиск объекта в списке.

Первый способ заключался в использовании созданной системы. Необходимо было сканировать объекты в любом порядке наведением сканера на QR-код. В приложении есть список, в который можно подглядеть, чтобы понять, какой объект еще не проверен. Система предоставляет удобный способ отображения предметов: если какой-либо объект был отсканирован, то он перемещается в конец списка. Таким образом, нет необходимо листать список приложения, если какой-либо объект был случайно пропущен. В самом верху всегда будут находиться неотмеченные предметы. После прохождения инвентаризации достаточно нажать кнопку «End check», после чего в документе описи будут отмечены фактическое наличие и фактическая стоимость.

Второй способ заключается в том, что человек держит в руках документ, в котором находится таблица объектов, и вручную отмечает объекты последовательно по списку.

Третий способ аналогичен первому за исключением того, что человек не идет последовательно по списку. Сотрудник смотрит на объекты вокруг себя и ищет их в списке, а только потом отмечает. Таким образом, человек может проверять группами по местоположению, а не ходить из одного угла кабинета в другой.

Каждый раз, когда человек отмечал своим способом 5, 10, 25, 50 и 100 объектов, фиксировалось время. Эксперимент для каждого способа проводился по 3 раза, результирующие значения взяты как среднее арифметическое.

|  |
| --- |
| Рисунок 21 – График с результатами эксперимента |
| Рисунок 22 – Таблица с результатами эксперимента |

Проведем анализ результатов эксперимента. Как можно увидеть, для 100 предметов приложение позволяет проводить инвентаризацию быстрее, чем последовательно по списку, в 2.5 раза. А также быстрее, чем сверка объектов по местоположению с поиском по списку, почти в 3 раза. Результаты предсказуемы, поскольку способ с последовательной разметкой плох тем, что приходится часто ходить с целью поиска предмета из списка. Метод сверки по местоположению плох тем, что приходится каждый раз проходить по всему огромному документу и искать конкретный объект, а потом в итоге все равно использовать первый способ, потому что некоторые предметы могли быть упущены.

Способ с приложением является наиболее быстрым и предпочтительным, а также наименее трудным. Самым тяжелым с точки зрения нагрузки на человека был способ с поиском объекта по местоположению, потому что приходилось искать конкретный один объект во всем документе.

В итоге результаты эксперимента показали, что созданное программное обеспечение действительно хорошо решает поставленные задачи, является, удобным, быстрым и эффективным.

# ЗАКЛЮЧЕНИЕ

В ходе выпускной квалификационной работы бакалавра был разработан программный комплекс, состоящий из серверной и клиентской частей. Серверная часть позволяет выполнять запросы на получение, добавление, изменение, удаление данных и авторизацию. Сторона сервера написана на языке программирования Java с использованием современного, надежного и развивающегося фреймворка Spring. Для поставленной задачи была спроектирована гибкая модель базы данных. Также при помощи архитектурного шаблона MVVM создана клиентская часть на базе операционной системы Android. В процессе разработки мобильного приложения использовался язык программирования Kotlin и современный подход для написания пользовательского интерфейса Jetpack Compose. Программное обеспечение позволяет пользователю удобно взаимодействовать с информацией об объектах инвентаризации и предоставляет дружественный пользовательский интерфейс по работе с QR-кодами.

Оценивая полноту решения поставленных задач, можно прийти к выводу, что задачи были решены в полной мере.

В качестве рекомендаций по развитию проекта можно предложить внедрение искусственного интеллекта на стадии добавления новых объектов. Так как на каждый предмет делается фотография, то можно реализовать функцию распознавания, которая по фотографии сразу определит класс объекта и предоставит соответствующую форму для заполнения полей. В качестве развития программного комплекса также можно предложить добавление новых сущностей для сканирования и более качественный процесс авторизации с применением методов криптографии.

Результаты проведенного эксперимента показали превосходство по скорости и простоте нового подхода проведения инвентаризации с QR-кодами. Представленное в работе решение превосходит традиционный подход проведения описи по скорости более, чем в 2.5 раза.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Мартин. Р Чистый код: анализ и рефакторинг. [Текст] – СПб.: Питер, 2022. – 464с.: ил. – ISBN 978-5-4461-0960-9
2. Макконнелл С. Совершенный код. Мастер-класс [Текст] / Пер. с англ. – СПб.: БХВ, 2022. – 896 стр.: ил. – ISBN 978-5-9909805-1-8
3. Крейг Уоллс, Spring в действии. 6-е изд./ пер. с англ. А. Н. Киселева. – М.: ДМКПресс, 2022. – 544 с.: ил.– ISBN 978-5-93700-112-2
4. Philipp Lackner, Youtube – канал Philipp Lackner [Электронный ресурс]. // Режим доступа: <https://www.youtube.com/@PhilippLackner> свободный (Дата обращения: 25.09.2023). Загл. с экрана.
5. Android Developers – Guide to app architecture [Электронный ресурс]. // Режим доступа: <https://developer.android.com/topic/architecture> свободный. (Дата обращения: 05.10.2023). Загл. с экрана.
6. Syed Ali Raza, Medium – XML vs Jetpack Compose: Choosing the Best UI Approach for Android [Электронный ресурс]. // Режима доступа: <https://aliraza112.medium.com/xml-vs-jetpack-compose-choosing-the-best-ui-approach-for-android-463d56aca983> свободный. (Дата обращения: 08.10.2023). Загл. с экрана.
7. Андрофишки, Дзен Статьи – Доля рынка Android и iOS в 2023 году [Электронный ресурс]. // Режим доступа: <https://dzen.ru/b/ZAG97vFccWym2waD> свободный. (Дата обращения: 02.03.2024). Загл. с экрана.
8. КонтурНорматив – Приказ Минфина РФ от 29.07.98 N 34Н [Электронный ресурс]. // Режим доступа: <https://normativ.kontur.ru/document?moduleId=1&documentId=312138> свободный. (Дата обращения: 03.03.2024). Загл. с экрана.
9. Дзен Статьи – Доля рынка Android и iOS в 2023 году [Электронный ресурс]. // Режим доступа: <https://dzen.ru/b/ZAG97vFccWym2waD> свободный. (Дата обращения: 02.03.2024). Загл. с экрана.
10. Старушенкова Е. Е., Палютин Р. Р., CyberLeninka – паттерн проектирования mvvm, как один из способов написания «чистого» кода в android-приложении на jetpack compose, [Электронный ресурс]. // Режим доступа: <https://cyberleninka.ru/article/n/pattern-proektirovaniya-mvvm-kak-odin-iz-sposobov-napisaniya-chistogo-koda-v-android-prilozhenii-na-jetpack-compose/viewer> свободный. (Дата обращения: 02.11.2023). Загл. с экрана.
11. Калюжный Е. Р., Красноусов В. М., Букреев Л. В, Зариковская Н. В.., CyberLeninka – технологии, используемые при реализации функционала мобильных приложений для платформы android, [Электронный ресурс]. // Режим доступа: <https://cyberleninka.ru/article/n/tehnologii-ispolzuemye-pri-realizatsii-funktsionala-mobilnyh-prilozheniy-dlya-platformy-android/viewer> свободный. (Дата обращения: 02.11.2023). Загл. с экрана.
12. Горячкин Б. С., Богданов Д. А., Щипицина К. В., CyberLeninka – эргономический анализ способов разработки интерфейса в android приложении, [Электронный ресурс]. // Режим доступа: <https://cyberleninka.ru/article/n/ergonomicheskiy-analiz-sposobov-razrabotki-interfeysa-v-android-prilozhenii/viewer> свободный. (Дата обращения: 02.11.2023). Загл. с экрана.
13. Алпатов Н С., Панов М. А., CyberLeninka – введение пропускной системы с использованием qr-кода, [Электронный ресурс]. // Режим доступа: <https://cyberleninka.ru/article/n/vvedenie-propusknoy-sistemy-s-ispolzovaniem-qr-koda/viewer> свободный. (Дата обращения: 03.11.2023). Загл. с экрана.
14. Здитовец А.Л., CyberLeninka – основные технологии и фреймворки для бекенд-разработки на java, [Электронный ресурс]. // Режим доступа: <https://cyberleninka.ru/article/n/osnovnye-tehnologii-i-freymvorki-dlya-bekend-razrabotki-na-java/viewer> свободный. (Дата обращения: 02.11.2023). Загл. с экрана.

# ПРИЛОЖЕНИЯ

## ПРИЛОЖЕНИЕ А

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | **@Service**  **public** **class** **ChairService** {  **private** **final** ChairRepository chairRepository;  **private** **final** ChairMapper chairMapper;  **@Autowired**  **public** **ChairService**(ChairRepository chairRepository, ChairMapper chairMapper) {  **this**.chairRepository = chairRepository;  **this**.chairMapper = chairMapper;  }  **public** List<ChairDTO> **getChairsByCabinetId**(Long cabinetId) {  var chairs = chairRepository.findChairsByCabinetId(cabinetId);  **if** (chairs.isEmpty()) {  **throw** **new** **ResourceNotFoundException**(STR."Chairs with cabinet id = \{cabinetId} not found");  }  **return** chairs  .stream()  .map(**chairMapper:**:toDto)  .toList();  }  **public** ChairDTO **saveChair**(ChairDTO chairDTO) {  var savedChair = chairRepository.save(chairMapper.toEntity(chairDTO));  **return** chairMapper.toDto(savedChair);  }  **public** ChairDTO **getChairById**(Long chairId) {  var chair = chairRepository.findById(chairId).orElseThrow(  () -> **new** ResourceNotFoundException(STR."Chair with id= \{chairId} not found")  );  **return** chairMapper.toDto(chair);  }  **public** **void** **deleteChairById**(Long chairId) {  var chair = chairRepository.findById(chairId).orElseThrow(  () -> **new** ResourceNotFoundException(STR."Chair with id= \{chairId} not found")  );  chairRepository.delete(chair);  }  } |

## ПРИЛОЖЕНИЕ Б

**@RestController**

**@RequestMapping**("/api/organizations/{orgId}/branches/{branchId}/buildings/{buildingId}/cabinets/{cabinetId}/chairs")

**public** **class** **ChairController** {

**private** **final** ChairService chairService;

**@Autowired**

**public** **ChairController**(ChairService chairService) {

**this**.chairService = chairService;

}

**@PostMapping**

**public** ResponseEntity<ChairDTO> **saveChair**(**@PathVariable**(required = **false**) Long orgId, **@PathVariable**(required = **false**) Long branchId, **@PathVariable**(required = **false**) Long buildingId, **@PathVariable**(required = **false**) Long cabinetId, **@RequestBody** ChairDTO chair) {

**return** **new** ResponseEntity<>(chairService.saveChair(chair), HttpStatus.CREATED);

}

**@PutMapping**

**public** ResponseEntity<ChairDTO> **updateChair**(**@PathVariable**(required = **false**) Long orgId, **@PathVariable**(required = **false**) Long branchId, **@PathVariable**(required = **false**) Long buildingId, **@PathVariable**(required = **false**) Long cabinetId, **@RequestBody** ChairDTO chair) {

**return** **new** ResponseEntity<>(chairService.saveChair(chair), HttpStatus.OK);

}

**@DeleteMapping**("/{chairId}")

**public** ResponseEntity<Void> **deleteChairById**(**@PathVariable**(required = **false**) Long orgId, **@PathVariable**(required = **false**) Long branchId, **@PathVariable**(required = **false**) Long buildingId, **@PathVariable**(required = **false**) Long cabinetId, **@PathVariable** Long chairId) {

chairService.deleteChairById(chairId);

**return** **new** ResponseEntity<>(HttpStatus.OK);

}

**@GetMapping**("/{chairId}")

**public** ResponseEntity<ChairDTO> **getChairById**(**@PathVariable**(required = **false**) Long orgId, **@PathVariable**(required = **false**) Long branchId, **@PathVariable**(required = **false**) Long buildingId, **@PathVariable**(required = **false**) Long cabinetId, **@PathVariable** Long chairId) {

**return** **new** ResponseEntity<>(chairService.getChairById(chairId), HttpStatus.OK);

}

}

## ПРИЛОЖЕНИЕ В

**public** **abstract** **class** **AbstractMapper**<E, D> **implements** Mapper<E, D> {

**@Autowired**

**protected** ModelMapper mapper;

**private** Class<E> entityClass;

**private** Class<D> dtoClass;

**protected** **AbstractMapper**(Class<E> entityClass, Class<D> dtoClass) {

**this**.entityClass = entityClass;

**this**.dtoClass = dtoClass;

}

**@Override**

**public** E **toEntity**(D dto) {

**return** Objects.isNull(dto)

? **null**

: mapper.map(dto, entityClass);

}

**@Override**

**public** D **toDto**(E entity) {

**return** Objects.isNull(entity)

? **null**

: mapper.map(entity, dtoClass);

}

**public** Converter<E, D> toDtoConverter() {

**return** context -> {

E source = context.getSource();

D destination = context.getDestination();

mapSpecificFieldsEntityToDto(source, destination);

**return** context.getDestination();

};

}

**public** Converter<D, E> toEntityConverter() {

**return** context -> {

D source = context.getSource();

E destination = context.getDestination();

mapSpecificFieldsDtoToEntity(source, destination);

**return** context.getDestination();

};

}

**protected** **void** **mapSpecificFieldsEntityToDto**(E source, D destination) {

}

**protected** **void** **mapSpecificFieldsDtoToEntity**(D source, E destination) {

}

}

## ПРИЛОЖЕНИЕ Г

**@Component**

**public** **final** **class** **BranchMapper** **extends** AbstractMapper<Branch, BranchDTO> {

**private** **final** CityRepository cityRepository;

**private** **final** OrganizationRepository organizationRepository;

**@Autowired**

BranchMapper(CityRepository cityRepository, OrganizationRepository organizationRepository) {

**super**(Branch.class, BranchDTO.class);

**this**.cityRepository = cityRepository;

**this**.organizationRepository = organizationRepository;

}

**@PostConstruct**

**public** **void** **setupMapper**() {

mapper.createTypeMap(Branch.class, BranchDTO.class)

.addMappings(

m -> {

m.skip(**BranchDTO:**:setCityId);

m.skip(**BranchDTO:**:setCityName);

m.skip(**BranchDTO:**:setOrganizationId);

}).setPostConverter(toDtoConverter());

mapper.createTypeMap(BranchDTO.class, Branch.class)

.addMappings(

m -> {

m.skip(**Branch:**:setCity);

m.skip(**Branch:**:setOrganization);

}).setPostConverter(toEntityConverter());

}

**@Override**

**protected** **void** **mapSpecificFieldsEntityToDto**(Branch source, BranchDTO destination) {

destination.setCityId(source.getCity().getId());

destination.setCityName(source.getCity().getName());

destination.setOrganizationId(source.getOrganization().getId());

}

**@Override**

**protected** **void** **mapSpecificFieldsDtoToEntity**(BranchDTO source, Branch destination) {

destination.setCity(

cityRepository.findById(source.getCityId())

.orElseThrow(() -> **new** ResourceNotFoundException(STR."City with id = \{source.getCityId()} not found")));

destination.setOrganization(organizationRepository.findById(source.getOrganizationId()).orElseThrow(

() -> **new** ResourceNotFoundException(STR."Organization with id = \{source.getOrganizationId()} not found")

));

}

}

## ПРИЛОЖЕНИЕ Д

**@RestControllerAdvice**

**@Slf4j**

**public** **class** **GlobalExceptionHandler** {

**@ExceptionHandler**

**@ResponseStatus**(HttpStatus.NOT\_FOUND)

**public** ResponseEntity<ErrorInfo> **catchResourceNotFoundException**(ResourceNotFoundException e) {

ErrorInfo errorInfo = **new** ErrorInfo(HttpStatus.NOT\_FOUND.value(), e.getMessage());

**return** **new** ResponseEntity<>(errorInfo, HttpStatusCode.valueOf(errorInfo.getStatus()));

}

**@ExceptionHandler**

**@ResponseStatus**(HttpStatus.FORBIDDEN)

**public** ResponseEntity<ErrorInfo> **catchWrongDataException**(WrongDataException e) {

ErrorInfo errorInfo = **new** ErrorInfo(HttpStatus.FORBIDDEN.value(), e.getMessage());

**return** **new** ResponseEntity<>(errorInfo, HttpStatusCode.valueOf(errorInfo.getStatus()));

}

**@ExceptionHandler**

**@ResponseStatus**(HttpStatus.FORBIDDEN)

**public** ResponseEntity<ErrorInfo> **catchAlreadyExistsException**(AlreadyExistsException e) {

ErrorInfo errorInfo = **new** ErrorInfo(HttpStatus.FORBIDDEN.value(), e.getMessage());

**return** **new** ResponseEntity<>(errorInfo, HttpStatusCode.valueOf(errorInfo.getStatus()));

}

}

## ПРИЛОЖЕНИЕ Е

**interface** InventarizedModel {

**var** id: Long

**var** image: Bitmap?

**var** inventoryNumber: String

**var** name: String

**var** cabinetId: Long

}

**interface** QRScannableData {

**fun** **getDatabaseTableName**(): DatabaseObjectTypes

**fun** **getDatabaseID**(): Long

}

data **class** **Chair**(

**override** **var** id: Long = **0L**,

**override** **var** image: Bitmap? = **null**,

**override** **var** inventoryNumber: String = "",

**override** **var** name: String = "",

**override** **var** cabinetId: Long = **0L**

) : InventarizedAndQRScannableModel, ToDTOMapper<ChairDTO> {

**override** **fun** **toDTO**(): ChairDTO {

**return** **ChairDTO**(

id,

image = image?.let { StaticConverters.fromBitmapToString(it) } ?: "",

inventoryNumber,

name,

cabinetId

)

}

**override** **fun** **getDatabaseTableName**(): DatabaseObjectTypes {

**return** DatabaseObjectTypes.CHAIR

}

**override** **fun** **getDatabaseID**(): Long {

**return** id

}

}

## ПРИЛОЖЕНИЕ Ж

**sealed** **class** **Resource**<T>(**val** data: T? = **null**, **val** message: String? = **null**) {

**class** **Loading**<T>(data: T? = **null**): Resource<T>(data)

**class** **Success**<T>(data: T?): Resource<T>(data)

**class** **Error**<T>(message: String, data: T? = **null**): Resource<T>(data, message)

}

//[M]odel, [D]TO, [A]PI, re[T]urning result, [AT] -- is returning type from api function

**interface** CRUDDeviceOperationsRepository<D : InventarizedDTO, A> {

**fun** **getAllDevicesInCabinet**(cabinetId: Long): Flow<Resource<List<InventarizedAndQRScannableModel>>>

**fun** **getDeviceById**(deviceId: Long): Flow<Resource<InventarizedAndQRScannableModel>>

**fun** **addDevice**(deviceDTO: D?): Flow<Resource<InventarizedAndQRScannableModel>>

**fun** **updateDevice**(deviceDTO: D): Flow<Resource<InventarizedAndQRScannableModel>>

**fun** **deleteDevice**(deviceId: Long): Flow<Resource<Unit>>

**fun** <T, AT> apiCall(

api: A,

context: Context,

bundleID: BundleID,

deviceDTO: D?,

apiCallFunction: suspend(

bundle: BundleID,

deviceDTO: D?

) -> Response<AT>,

getRequestBodyAndConvertInModel: (Response<AT>) -> T

): Flow<Resource<T>> = flow {

emit(Resource.Loading())

**try** {

**val** result = tryToMakeAPICall(

api,

context,

bundleID,

deviceDTO,

apiCallFunction,

getRequestBodyAndConvertInModel

)

emit(Resource.Success(result))

} **catch** (e: RemoteServerRequestException) {

Log.e("ERROR", e.message ?: "Unknown error")

emit(Resource.Error(message = e.message ?: "Unknown error"))

}

}

//not needed parameters could be omitted

suspend **fun** <T, AT> tryToMakeAPICall(

api: A,

context: Context,

bundleID: BundleID,

deviceDTO: D?,

apiCallFunction: suspend (

bundleID: BundleID,

deviceDTO: D?

) -> Response<AT>,

getRequestBodyAndConvertInModel: (Response<AT>) -> T

): T {

**val** message: String

**try** {

**val** response = apiCallFunction(bundleID, deviceDTO)

**return** **getRequestBodyAndConvertInModel**(response)

} **catch** (e: HttpException) {

message = ExceptionAndErrorParsers.getErrorMessageFromException(e)

**throw** **RemoteServerRequestException**(message)

} **catch** (e: IOException) {

message = context.getString(R.string.server\_connection\_error)

**throw** **RemoteServerRequestException**(message)

}

}

}

## ПРИЛОЖЕНИЕ И

**fun** **onEvent**(event: QRScannerEvent) {

**when** (event) {

**is** QRScannerEvent.OnAddObjectInList -> {

...

}

**is** QRScannerEvent.OnAddScannedGroup -> {

...

}

**is** QRScannerEvent.OnScanQRCode -> {

onScanQRCode(event.scannedData)

}

**is** QRScannerEvent.OnAddNameForScannedGroup -> {

...

}

**is** QRScannerEvent.ClearListOfScannedObjects -> {

...

}

**is** QRScannerEvent.OnGoToScannedGroupsWindow -> {

sendUiEvent(UIScannerEvent.Navigate(context.resources.getString(R.string.list\_of\_groups\_route)))

}

**is** QRScannerEvent.OnDeleteDeviceFromServerClicked -> {

sendUiEvent(

UIScannerEvent.ShowMessagedDialogWindow(

message = "Are you sure you want to delete device: " +

"\"${\_previewDataFromQRState.value.scannedDataInfo?.name}\" from the server?",

onDeclineAction = {},

onAcceptAction = {

deleteDeviceFromServer(\_previewDataFromQRState.value.scannedDataInfo **as** QRScannableData)

},

dialogTitle = "Delete device",

icon = Icons.Default.Dangerous

)

)

}

}

}

**private** **fun** **onScanQRCode**(scanResult: String) {

**if** (scanResult.isBlank() || prevScanString == scanResult) {

**return**

}

setDeduplicateStringAndDelayForClear(scanResult)

**try** {

**val** scannedObject = converter.fromJsonToScannedTableNameAndId(scanResult)

scanJob?.cancel()

insertScannedDataInStateIfPossible(scannedObject)

} **catch** (e: JsonSyntaxException) {

showQRCodeFormatError(e)

}

}

**private** **fun** **setDeduplicateStringAndDelayForClear**(scanResult: String) {

prevScanString = scanResult

deleteDuplicateQRCodeStringJob?.cancel()

deleteDuplicateQRCodeStringJob = CoroutineScope(Dispatchers.Default).launch {

delay(timeForDuplicateQRCodesResistInMs)

prevScanString = ""

}

}

**private** **fun** **insertScannedDataInStateIfPossible**(scannedObject: DeviceInfoInQRCodeRepresenter?) {

scanJob = viewModelScope.launch {

scannedObject?.let { scannedObj ->

**try** {

getDataFromQRCodeUseCase =

useCaseGetQRCodeFactory.createUseCase(scannedObj.tableName)

} **catch** (error: ClassNotFoundException) {

Log.e(LOG\_TAG\_NAMES.ERROR\_TAG, error.message ?: "Unknown error")

**val** errorMsg = error.message ?: "Unknown error"

setPreviewObjectStateInfo(Resource.Error(message = errorMsg, Unknown(errorMsg)))

**return**@launch

}

getDataFromQRCodeUseCase(scannedObj.id).onEach { result ->

setPreviewObjectStateInfo(result)

}.launchIn(**this**)

} ?: Log.e(LOG\_TAG\_NAMES.ERROR\_TAG, "Error with scannedObject convertion. No id there")

}

}

**private** **fun** **setPreviewObjectStateInfo**(data: Resource<InventarizedAndQRScannableModel>) {

**when** (data) {

**is** Resource.Loading -> {

setDataWithStatus(data, **true**)

}

**is** Resource.Error -> {

setErrorStatusAndSendSnackbarEvent(data.message)

}

**is** Resource.Success -> {

setPlaceByDeviceAndUser(data.data)

setDataWithStatus(data, **false**)

}

}

}

## ПРИЛОЖЕНИЕ К

//From ViewModel to View

**sealed** **class** **UIScannerEvent** {

data object PopBackStack: UIScannerEvent() //Navigate back to prev screen

data **class** **Navigate**(**val** route: String): UIScannerEvent()

data **class** **ShowSnackBar**(**val** message: String) : UIScannerEvent()

data **class** **ShowMessagedDialogWindow**(

**val** message: String,

**val** onDeclineAction: () -> Unit,

**val** onAcceptAction: () -> Unit,

**val** dialogTitle: String,

**val** icon: ImageVector

) : UIScannerEvent()

data object ShowScannedGroupNameDialogWindow : UIScannerEvent()

}

//From View to ViewModel

**sealed** **class** **QRScannerEvent** {

data **class** **OnAddObjectInList**(

**val** scannableObject: InventarizedAndQRScannableModel,

**val** addEvenIfDuplicate: Boolean

) : QRScannerEvent()

data **class** **OnScanQRCode**(**val** scannedData: String) : QRScannerEvent()

data object OnAddScannedGroup : QRScannerEvent()

data **class** **OnAddNameForScannedGroup**(**val** groupName: String) : QRScannerEvent()

data object ClearListOfScannedObjects : QRScannerEvent()

data object OnGoToScannedGroupsWindow : QRScannerEvent()

data **class** **OnDeleteDeviceFromServerClicked**(**val** qrScannableData: QRScannableData) :

QRScannerEvent()

}

## ПРИЛОЖЕНИЕ Л

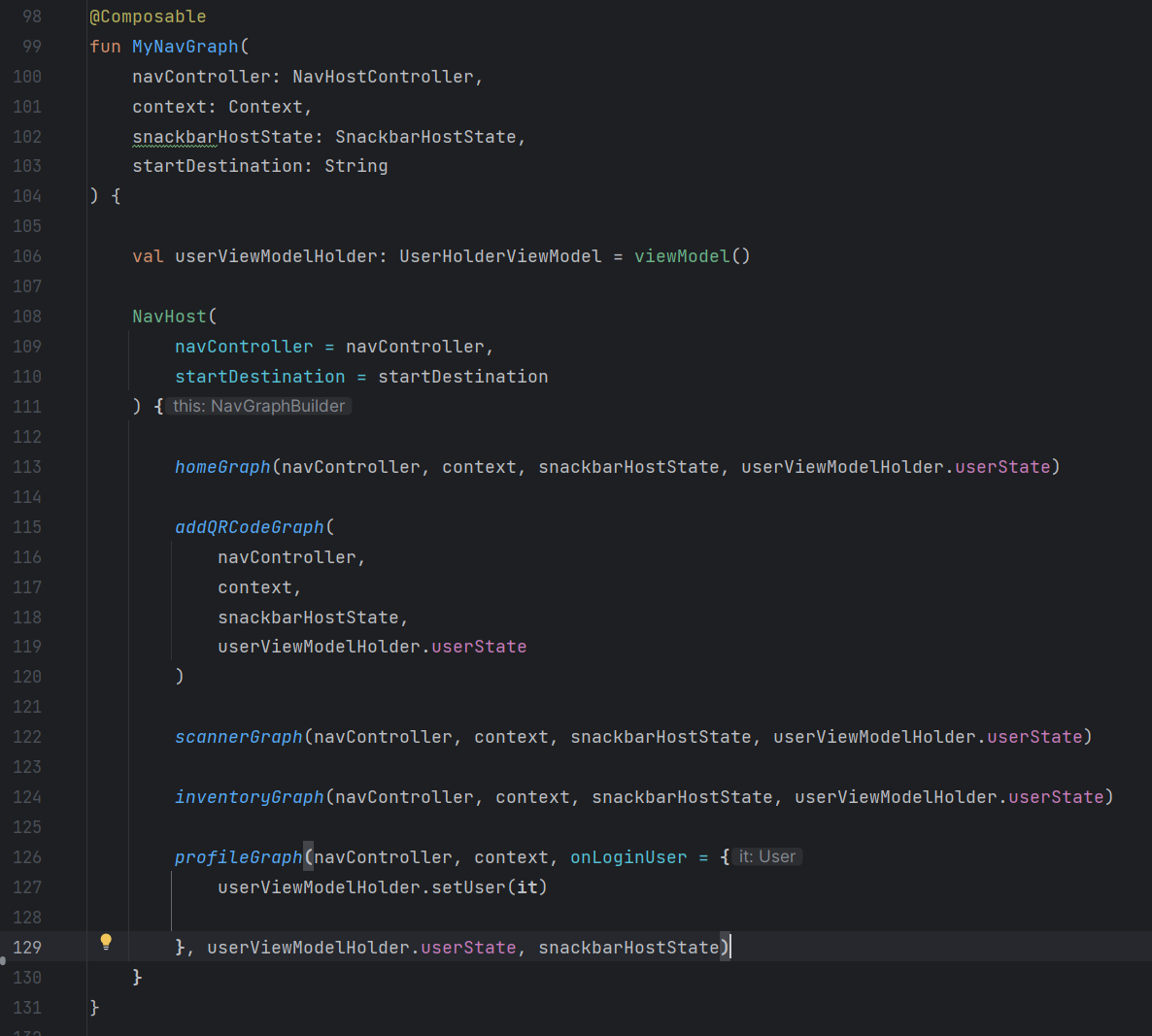


Рисунок 23 – Общая структура графа

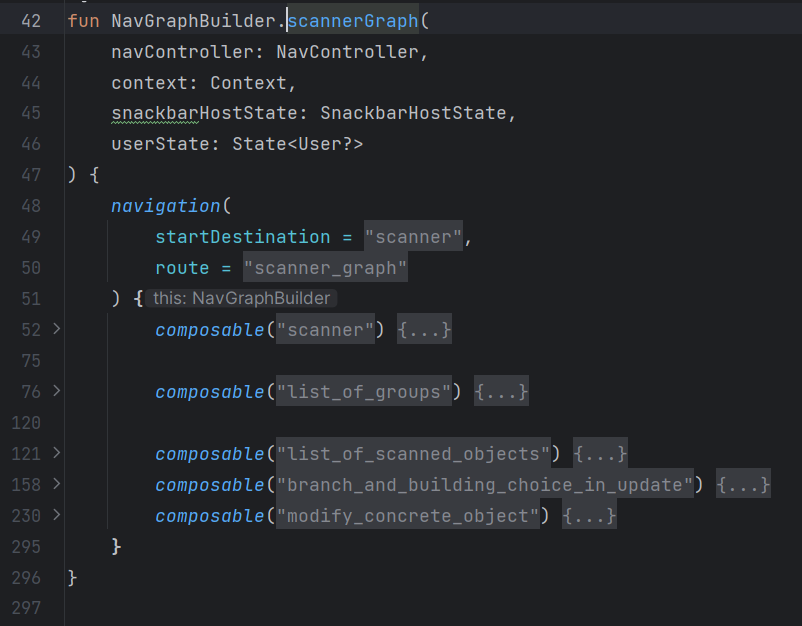


Рисунок 24 – Ветвь сканирования

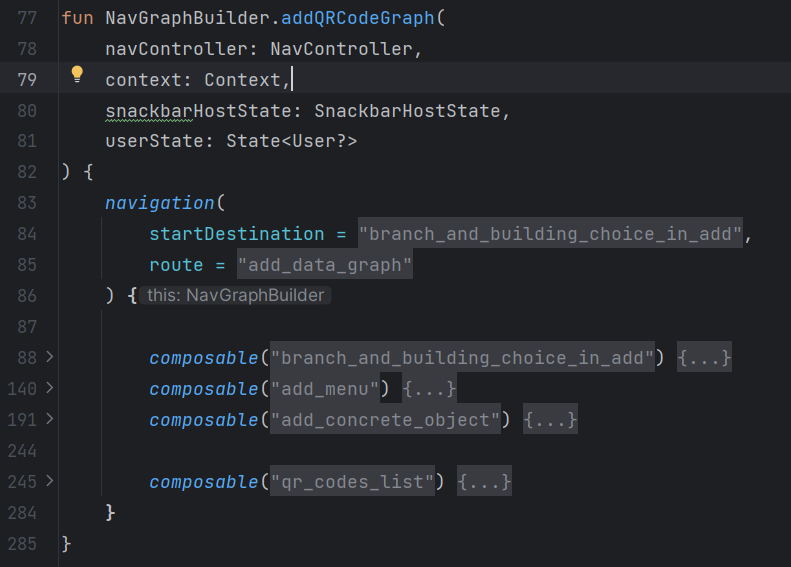


Рисунок 25 – Ветвь добавления новых объектов

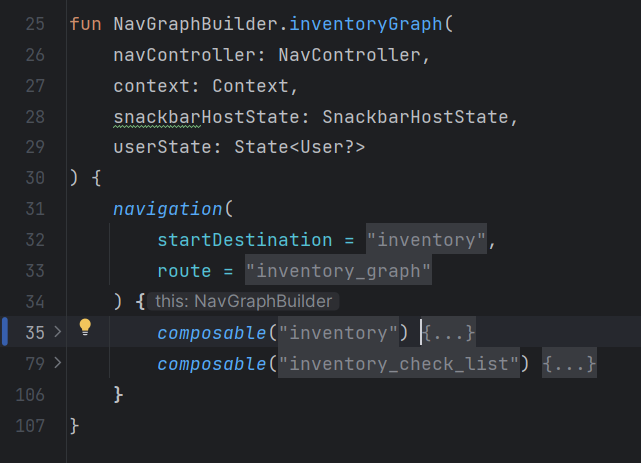


Рисунок 26 – Ветвь проведения инвентаризации

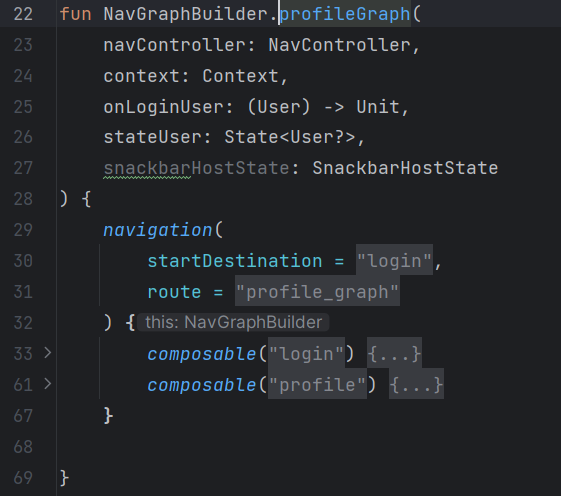


Рисунок 27 – Ветвь профиля и авторизации

## ПРИЛОЖЕНИЕ М

@Composable

inline **fun** <reified T : ViewModel> NavBackStackEntry.sharedScannedObjectsListInGroupViewModel(

scannedGroup: State<ScannedGroup>,

navController: NavController,

userAndPlaceBundle: UserAndPlaceBundle

): T {

**val** navGraphRoute = destination.parent?.route ?: **return** viewModel()

**val** parentEntry = remember(**this**) {

navController.getBackStackEntry(navGraphRoute)

}

**val** factory = EntryPointAccessors.fromActivity(

LocalContext.current **as** Activity,

ScannedDataViewModelFactoryProvider::class.java

).scannedObjectsListViewModelFactory()

**return** **viewModel**(

viewModelStoreOwner = parentEntry,

factory = ScannedObjectsListViewModel.provideFactory(

factory,

scannedGroup,

userAndPlaceBundle

)

)

}

## ПРИЛОЖЕНИЕ Н

@Module

**@InstallIn**(SingletonComponent::class)

object UseCaseModule {

@Provides

@Singleton

**fun** **provideQRCodeScannerUseCasesFactory**(

qrCodeScannerUseCases: DataGetterUseCases,

@ApplicationContext context: Context

): GetObjectFromServerUseCaseFactory {

**return** **GetObjectFromServerUseCaseFactory**(qrCodeScannerUseCases, context)

}

@Provides

@Singleton

**fun** **getPlaceUseCases**(

getOrganizationUseCase: GetOrganizationUseCase,

getBranchUseCase: GetBranchUseCase,

getBuildingUseCase: GetBuildingUseCase,

getCabinetUseCase: GetCabinetUseCase

): GetPlaceUseCases {

**return** **GetPlaceUseCases**(

getOrganizationUseCase,

getBranchUseCase,

getBuildingUseCase,

getCabinetUseCase

)

}

@Provides

@Singleton

**fun** **provideQRCodeScannerUseCases**(

getChairUseCase: GetChairUseCase,

getDeskUseCase: GetDeskUseCase,

getKeyboardUseCase: GetKeyboardUseCase,

getMonitorUseCase: GetMonitorUseCase,

getSystemUnitUseCase: GetSystemUnitUseCase,

getProjectorUseCase: GetProjectorUseCase

): DataGetterUseCases {

**return** **DataGetterUseCases**(

getChairUseCase,

getDeskUseCase,

getProjectorUseCase,

getMonitorUseCase,

getSystemUnitUseCase,

getKeyboardUseCase

)

}

@Provides

@Singleton

**fun** **provideDeleteDeviceOnServerUserCases**(

deleteChairUseCase: DeleteChairUseCase,

deleteDeskUseCase: DeleteDeskUseCase,

deleteKeyboardUseCase: DeleteKeyboardUseCase,

deleteProjectorUseCase: DeleteProjectorUseCase,

deleteSystemUnitUseCase: DeleteSystemUnitUseCase,

deleteMonitorUseCase: DeleteMonitorUseCase

): DeleteObjectOnServerUseCases {

**return** **DeleteObjectOnServerUseCases**(

deleteChairUseCase,

deleteDeskUseCase,

deleteKeyboardUseCase,

deleteProjectorUseCase,

deleteSystemUnitUseCase,

deleteMonitorUseCase

)

}

@Provides

@Singleton

**fun** **provideDeleteDeviceOnServerUseCasesFactory**(

deleteDeviceOnServerUseCases: DeleteObjectOnServerUseCases,

@ApplicationContext context: Context

): DeleteObjectOnServerUseCaseFactory {

**return** **DeleteObjectOnServerUseCaseFactory**(deleteDeviceOnServerUseCases, context)

}

}

@Module

**@InstallIn**(SingletonComponent::class)

object ChairModule {

@Provides

@Singleton

**fun** **provideGetChairsUseCase**(chairRepository: ChairRepository): GetAllChairsInCabinetUseCase {

**return** **GetAllChairsInCabinetUseCase**(chairRepository)

}

@Provides

@Singleton

**fun** **provideGetChairUseCase**(chairRepository: ChairRepository): GetChairUseCase {

**return** **GetChairUseCase**(chairRepository)

}

@Provides

@Singleton

**fun** **provideAddChairUseCase**(chairRepository: ChairRepository): AddChairUseCase {

**return** **AddChairUseCase**(chairRepository)

}

@Provides

@Singleton

**fun** **provideUpdateChairUseCase**(chairRepository: ChairRepository): UpdateChairUseCase {

**return** **UpdateChairUseCase**(chairRepository)

}

@Provides

@Singleton

**fun** **provideDeleteChairUseCase**(chairRepository: ChairRepository): DeleteChairUseCase {

**return** **DeleteChairUseCase**(chairRepository)

}

@Provides

@Singleton

**fun** **provideChairRepository**(

chairAPI: ChairAPI,

@ApplicationContext context: Context

): ChairRepository {

**return** **ChairRepositoryImpl**(chairAPI, context)

}

@Provides

@Singleton

**fun** **provideChairApi**(okHttpClient: OkHttpClient): ChairAPI {

**return** Retrofit.Builder()

.baseUrl(ApiInfo.BASE\_URL)

.addConverterFactory(GsonConverterFactory.create())

.client(okHttpClient)

.build()

.create(ChairAPI::class.java)

}

}

## ПРИЛОЖЕНИЕ П

|  |  |
| --- | --- |
| Рисунок 28 – Экран авторизации | Рисунок 29 – Требование авторизации |

## ПРИЛОЖЕНИЕ Р

**abstract** **class** **BaseAddObjectViewModel**(

**private** **val** context: Context,

**private** **val** converter: Converters,

**private** **val** userAndPlaceBundle: UserAndPlaceBundle

) : ViewModel() {

**private** **val** \_eventFlow = Channel<UIAddNewObjectEvent>()

**val** eventFlow = \_eventFlow.receiveAsFlow()

**private** **fun** **createQRCode**(qrScannableData: InventarizedAndQRScannableModel): ImageBitmap {

**val** convertedJsonFromString = converter.toJsonFromQRScannableData(

qrScannableData,

userAndPlaceBundle.user.organizationId

)

**val** bitmap = QRCodeGenerator.encodeAsBitmap(convertedJsonFromString, **250**, **250**)

**return** bitmap.asImageBitmap()

}

**abstract** **fun** **addObjectInDatabaseClicked**(

onAddObjectClicked: (QRCodeStickerInfo) -> Unit = {},

afterUpdateAction: () -> Unit = {}

)

**protected** **fun** **sendUiEvent**(event: UIAddNewObjectEvent) {

viewModelScope.launch {

\_eventFlow.send(event)

}

}

**private** **fun** **isBasicFieldsInsertedCorrectly**(device: InventarizedModel): Boolean {

**return** device.image != **null** &&

device.name.isNotBlank() &&

device.cabinetId != **0L** &&

device.inventoryNumber.isNotBlank()

}

**protected** **fun** **isSpecificFieldInsertedCorrectly**(): Boolean {

**return** **true**

}

**protected** **fun** **isAllNeededFieldsInsertedCorrectly**(device: InventarizedModel): Boolean {

**return** **isSpecificFieldInsertedCorrectly**() && isBasicFieldsInsertedCorrectly(device)

}

**abstract** **fun** **isAllNeededFieldsInsertedCorrectly**(): Boolean

**fun** **onEvent**(event: AddNewObjectEvent) {

**when** (event) {

**is** AddNewObjectEvent.OnImageChanged -> {

setNewImage(event.image)

}

**is** AddNewObjectEvent.OnInventoryNumberChanged -> {

setNewInventoryNumber(event.inventoryNumber)

}

**is** AddNewObjectEvent.OnNameChanged -> {

setNewName(event.name)

}

**is** AddNewObjectEvent.OnCabinetChanged -> {

setNewCabinetId(event.cabinetId)

}

}

}

**protected** **fun** **scaleImage**(image: Bitmap): Bitmap {

**val** aspectRatio: Double = image.width.toDouble() / image.height.toDouble()

**val** width = ImageConstants.RESIZED\_WIDTH

**val** height = (width / aspectRatio).roundToInt()

**return** Bitmap.createScaledBitmap(

image, width, height, **false**

)

}

**protected** **abstract** **fun** **setNewImage**(image: Bitmap?)

**protected** **abstract** **fun** **setNewName**(name: String)

**protected** **abstract** **fun** **setNewCabinetId**(cabinetId: Long)

**protected** **abstract** **fun** **setNewInventoryNumber**(invNumber: String)

**protected** **fun** <M : InventarizedAndQRScannableModel> makeActionWithResourceResult(

statusWithState: Resource<M>,

deviceState: MutableState<BaseDeviceState>,

onAddObjectClicked: (QRCodeStickerInfo) -> Unit = {},

afterUpdateAction: () -> Unit = {},

qrCodeStickerInfo: QRCodeStickerInfo? = **null**

) {

**when** (statusWithState) {

**is** Resource.Error -> {

deviceState.value = deviceState.value.stateCopy(

deviceState.value.deviceState,

**false**

)

sendUiEvent(

UIAddNewObjectEvent.ShowSnackBar(

statusWithState.message ?: "Unknown error"

)

)

}

**is** Resource.Loading -> {

deviceState.value = deviceState.value.stateCopy(

deviceState.value.deviceState,

**true**

)

}

**is** Resource.Success -> {

deviceState.value.deviceState.value.id = statusWithState.data?.id ?: **0**

// Not need for update

qrCodeStickerInfo?.let { existingQRCodeSticker ->

setQRStickerInfo(

statusWithState.data **as** InventarizedAndQRScannableModel,

existingQRCodeSticker

)

onAddObjectClicked(existingQRCodeSticker)

} ?: afterUpdateAction()

// Not need for update

deviceState.value = deviceState.value.stateCopy(

deviceState.value.deviceState,

**false**

)

Log.i(LOG\_TAG\_NAMES.INFO\_TAG, "UPDATE")

qrCodeStickerInfo?.let {

sendUiEvent(UIAddNewObjectEvent.Navigate(context.resources.getString(R.string.menu\_add\_route)))

}

?: sendUiEvent(UIAddNewObjectEvent.Navigate(context.resources.getString(R.string.list\_of\_scanned\_objects)))

}

}

}

**protected** **abstract** **fun** **getEssentialNameForQRCodeSticker**(): String

**private** **fun** **setQRStickerInfo**(

device: InventarizedAndQRScannableModel,

qrCodeStickerInfo: QRCodeStickerInfo

) {

device.let {

qrCodeStickerInfo.qrCode = createQRCode(device)

qrCodeStickerInfo.essentialName = getEssentialNameForQRCodeSticker()

qrCodeStickerInfo.inventoryNumber = device.inventoryNumber

qrCodeStickerInfo.databaseObjectTypes = device.getDatabaseTableName()

}

}

}

**class** **AddDeskViewModel** @AssistedInject **constructor**(

@Assisted **private** **val** userAndPlaceBundle: UserAndPlaceBundle,

@ApplicationContext **private** **val** context: Context,

converters: Converters,

**private** **val** addDeskUseCase: AddDeskUseCase,

@Assisted **private** **val** deskForUpdate: InventarizedAndQRScannableModel?,

**private** **val** updateDeskUseCase: UpdateDeskUseCase

) : BaseAddObjectViewModel(context, converters, userAndPlaceBundle) {

**private** **val** \_deskState = mutableStateOf(

deskForUpdate ?: Desk(

cabinetId = userAndPlaceBundle.cabinet.id

)

) //work with this here is more convenient

**private** **val** \_deskStateWithLoadingStatus = mutableStateOf<BaseDeviceState>(

DeskUIState(

\_deskState, **false**

)

)

**val** deskStateWithLoadingStatus: State<BaseDeviceState> = \_deskStateWithLoadingStatus

@AssistedFactory

**interface** Factory {

**fun** **create**(

userAndPlaceBundle: UserAndPlaceBundle,

deviceForUpdate: InventarizedAndQRScannableModel?

): AddDeskViewModel

}

companion object {

@Suppress("UNCHECKED\_CAST")

**fun** **provideFactory**(

assistedFactory: Factory,

userAndPlaceBundle: UserAndPlaceBundle,

deviceForUpdate: InventarizedAndQRScannableModel?

): ViewModelProvider.Factory = object : ViewModelProvider.Factory {

**override** **fun** <T : ViewModel> create(modelClass: Class<T>): T {

**return** assistedFactory.create(userAndPlaceBundle, deviceForUpdate) **as** T

}

}

}

**override** **fun** **addObjectInDatabaseClicked**(

onAddObjectClicked: (QRCodeStickerInfo) -> Unit,

afterUpdateAction: () -> Unit

) {

**val** desk = (\_deskState.value **as** Desk).toDTO()

viewModelScope.launch {

deskForUpdate?.let {

Log.i(LOG\_TAG\_NAMES.INFO\_TAG, "UPDATE API")

updateDeskUseCase(desk).onEach { statusWithState ->

makeActionWithResourceResult(

statusWithState = statusWithState,

deviceState = \_deskStateWithLoadingStatus,

afterUpdateAction = afterUpdateAction,

)

}.launchIn(**this**)

} ?: addDeskUseCase(desk).onEach { statusWithState ->

makeActionWithResourceResult(

statusWithState = statusWithState,

deviceState = \_deskStateWithLoadingStatus,

onAddObjectClicked = onAddObjectClicked,

qrCodeStickerInfo = QRCodeStickerInfo()

)

}.launchIn(**this**)

}

}

**override** **fun** **isAllNeededFieldsInsertedCorrectly**(): Boolean {

**return** **isAllNeededFieldsInsertedCorrectly**(\_deskState.value)

}

**override** **fun** **setNewImage**(image: Bitmap?) {

image?.let {

**val** rescaledImage = scaleImage(it)

\_deskState.value = Desk(

id = \_deskState.value.id,

image = rescaledImage,

inventoryNumber = \_deskState.value.inventoryNumber,

name = \_deskState.value.name,

cabinetId = \_deskState.value.cabinetId

)

}

}

**override** **fun** **setNewName**(name: String) {

\_deskState.value = Desk(

id = \_deskState.value.id,

image = \_deskState.value.image,

inventoryNumber = \_deskState.value.inventoryNumber,

name = name,

cabinetId = \_deskState.value.cabinetId

)

}

**override** **fun** **setNewCabinetId**(cabinetId: Long) {

\_deskState.value = Desk(

id = \_deskState.value.id,

image = \_deskState.value.image,

inventoryNumber = \_deskState.value.inventoryNumber,

name = \_deskState.value.name,

cabinetId = cabinetId

)

}

**override** **fun** **setNewInventoryNumber**(invNumber: String) {

\_deskState.value = Desk(

id = \_deskState.value.id,

image = \_deskState.value.image,

inventoryNumber = invNumber,

name = \_deskState.value.name,

cabinetId = \_deskState.value.cabinetId

)

}

**override** **fun** **getEssentialNameForQRCodeSticker**(): String {

**return** \_deskState.value.name

}

}

## ПРИЛОЖЕНИЕ С

|  |  |
| --- | --- |
| Рисунок 30 – Экран выбора кабинета | Рисунок 31 – Экран выбора объекта для добавления |
| Рисунок 32 – Заполнение полей на добавление | Рисунок 33 – Заполнение полей на добавление |

|  |  |
| --- | --- |
| Рисунок 34 – Отображение количества добавленных QR-кодов | Рисунок 35 – Выбор размера QR-кодов |
| Рисунок 36 – Предоставление доступа к каталогу для сохранения | Рисунок 37 – Вид файла на мобильном устройстве |

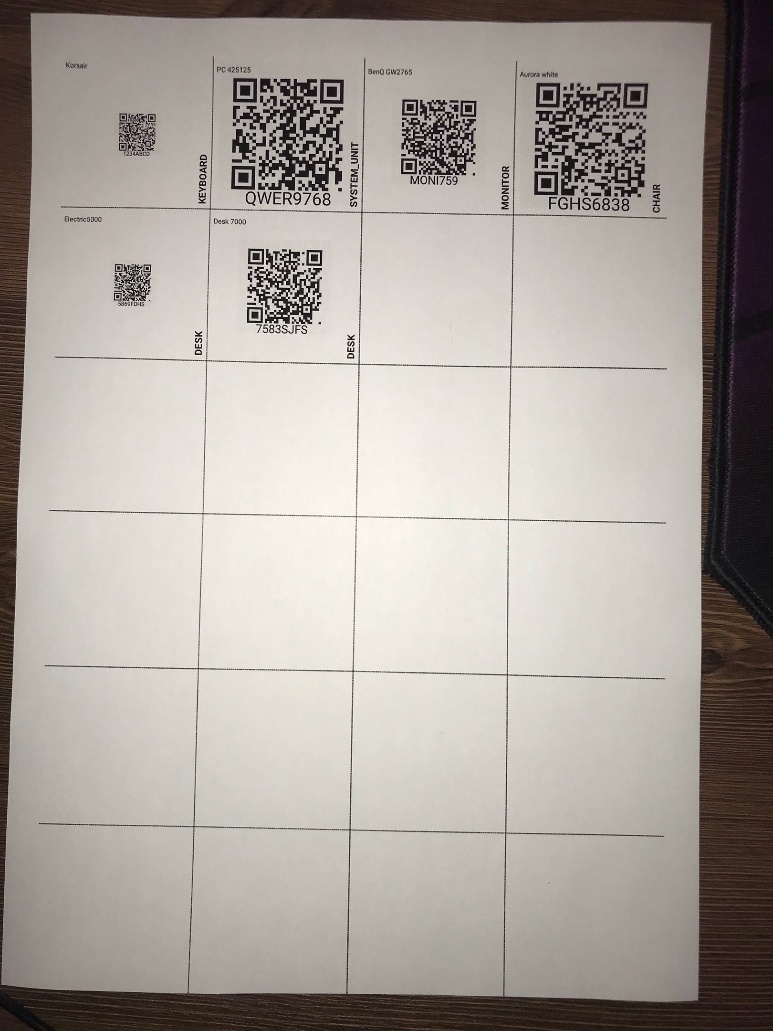


Рисунок 38 – Распечатанный документ с QR-кодами

## ПРИЛОЖЕНИЕ Т

AndroidView(

modifier = Modifier

.scale(**0.8f**)

.weight(**1f**)

.align(Alignment.CenterHorizontally),

factory = { context ->

**val** previewView = PreviewView(context)

**val** preview = Preview.Builder().build()

**val** selector = CameraSelector.Builder()

.requireLensFacing(CameraSelector.LENS\_FACING\_BACK)

.build()

preview.setSurfaceProvider(previewView.surfaceProvider)

**val** resolutionSelector = ResolutionSelector.Builder()

.setResolutionStrategy(ResolutionStrategy.HIGHEST\_AVAILABLE\_STRATEGY)

.build()

**val** imageAnalysis = ImageAnalysis.Builder()

.setResolutionSelector(resolutionSelector)

.setBackpressureStrategy(ImageAnalysis.STRATEGY\_KEEP\_ONLY\_LATEST)

.build()

imageAnalysis.setAnalyzer(

ContextCompat.getMainExecutor(context),

QRCodeAnalyzer { scannedString ->

viewModel.onEvent(QRScannerEvent.OnScanQRCode(scannedString))

}

)

**try** {

cameraProviderFuture.**get**().bindToLifecycle(

lifecycleOwner,

selector,

preview,

imageAnalysis

)

} **catch** (e: Exception) {

e.printStackTrace()

}

previewView

},

)

**class** **QRCodeAnalyzer**(

**private** **val** onQrCodeScanned: (String) -> Unit

): ImageAnalysis.Analyzer {

**private** **val** supportedImageFormats = listOf(

ImageFormat.YUV\_420\_888,

ImageFormat.YUV\_422\_888,

ImageFormat.YUV\_444\_888

)

**override** **fun** **analyze**(image: ImageProxy) {

**if**(image.format **in** supportedImageFormats){

**val** bytes = image.planes.first().buffer.toByteArray()

**val** source = PlanarYUVLuminanceSource(

bytes,

image.width,

image.height,

**0**,

**0**,

image.width,

image.height,

**false**

)

**val** binaryBitMap = BinaryBitmap(HybridBinarizer(source)) //contains info about scanned qr

**try** {

**val** result = MultiFormatReader().apply {

setHints(

mapOf(

DecodeHintType.POSSIBLE\_FORMATS to arrayListOf(

BarcodeFormat.QR\_CODE

)

)

)

}.decode(binaryBitMap)

onQrCodeScanned(result.text)

}

**catch** (e: Exception){

onQrCodeScanned("")

}

**finally** {

image.close()

}

}

}

**private** **fun** ByteBuffer.toByteArray() : ByteArray{

rewind()

**return** **ByteArray**(remaining()).also {

**get**(it)

}

}

}

## ПРИЛОЖЕНИЕ У

|  |  |
| --- | --- |
| Рисунок 39 – Сканирование QR-кода клавиатуры | Рисунок 40 – Сканирование QR-кода стула |
| Рисунок 41 – Сканирование QR-кода стола | Рисунок 42 – Сканирование QR-кода монитора |

## ПРИЛОЖЕНИЕ Ф

|  |  |
| --- | --- |
| Рисунок 43 – До смены названия монитора | Рисунок 44 – После смены названия монитора |
| Рисунок 45 – Экран модификации полей | Рисунок 46 – Экран модификации полей |

## ПРИЛОЖЕНИЕ Х

|  |  |
| --- | --- |
| Рисунок 47 – Сообщение перед списанием объекта | Рисунок 48 – Результат сканирования после списания |

## ПРИЛОЖЕНИЕ Ц

|  |  |
| --- | --- |
| Рисунок 49 – Создание каталога | Рисунок 50 – Отображение каталогов |
| Рисунок 51 – Иконка удаления при удержании иконки | Рисунок 52 – Сообщение перед удаление каталога |

## ПРИЛОЖЕНИЕ Ш

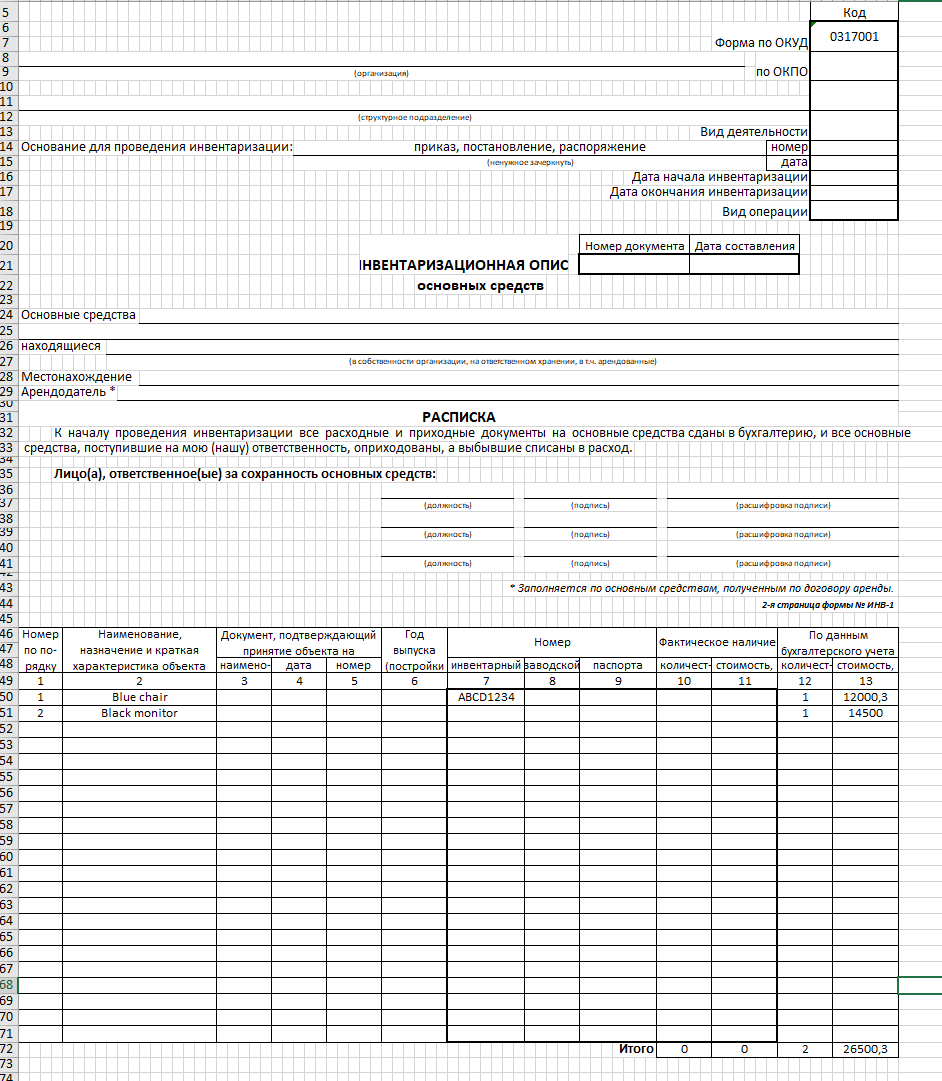


Рисунок 53 – Пример формы ИНВ-1

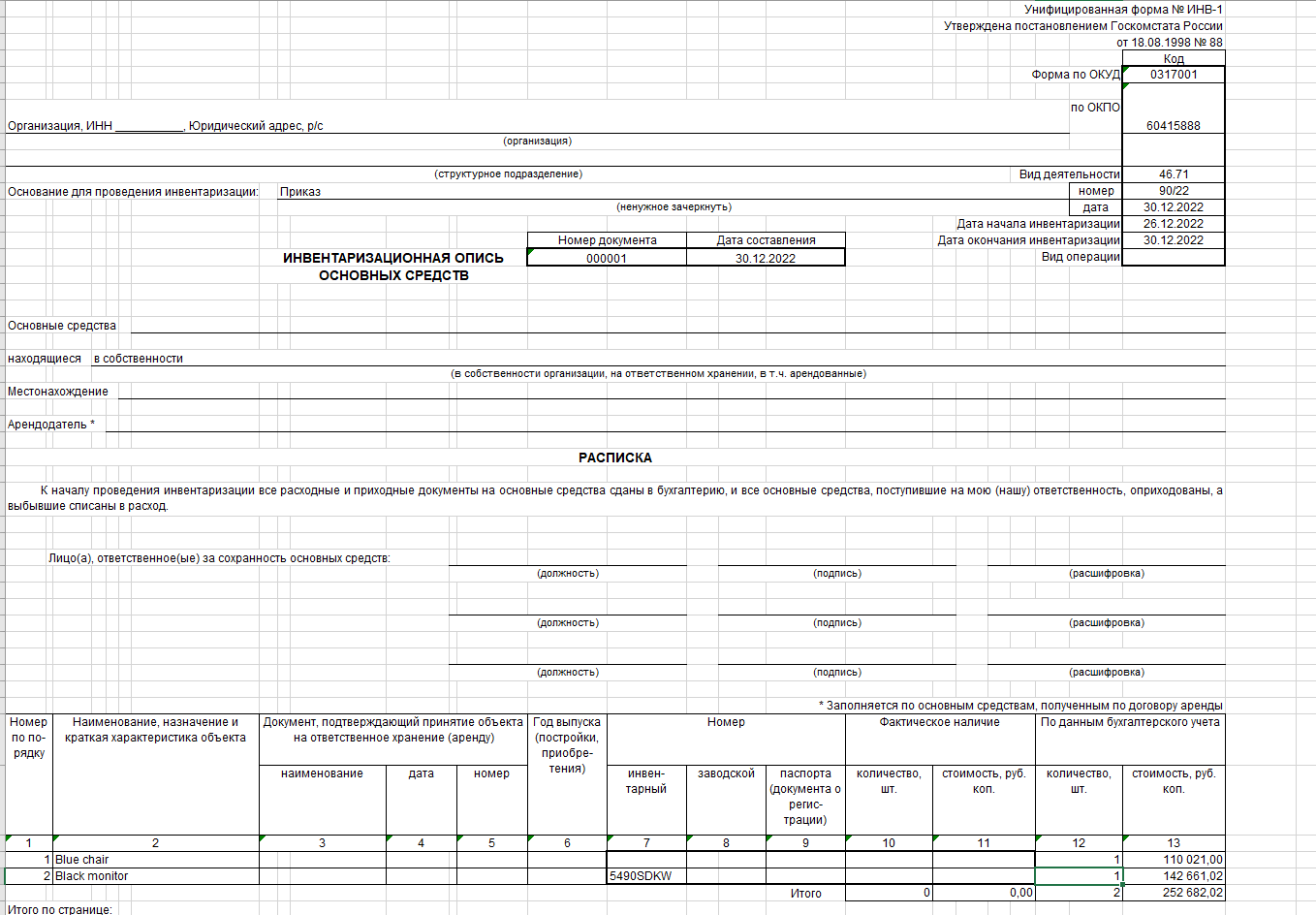


Рисунок 54 – Пример формы ИНВ-1



Рисунок 55 – Пример формы ИНВ-1

## ПРИЛОЖЕНИЕ Щ

**class** **InventarizedINV\_1FileParser**(

**private** **val** valueOfTableColumnsInDocument: Int = **13**,

**private** **val** mapWithColumnNumberToNameBijection: Map<Int, InventarizedColumnNames> = mapOf(

**1** to InventarizedColumnNames.ORD\_NUMBER,

**2** to InventarizedColumnNames.OBJ\_NAME,

**3** to InventarizedColumnNames.DOC\_RENT\_NAME,

**4** to InventarizedColumnNames.DOC\_RENT\_DATE,

**5** to InventarizedColumnNames.DOC\_RENT\_NUMBER,

**6** to InventarizedColumnNames.RELEASE\_YEAR,

**7** to InventarizedColumnNames.INV\_NUMBER,

**8** to InventarizedColumnNames.FACTORY\_NUMBER,

**9** to InventarizedColumnNames.PASSPORT\_NUMBER,

**10** to InventarizedColumnNames.FACT\_QUANTITY,

**11** to InventarizedColumnNames.FACT\_PRICE,

**12** to InventarizedColumnNames.ACCOUNTANT\_QUANTITY,

**13** to InventarizedColumnNames.ACCOUNTANT\_PRICE,

),

**private** **val** listOfDocumentColumns: MutableList<InventarizedColumn> = MutableList(

valueOfTableColumnsInDocument

) { InventarizedColumn() },

**val** listOfObjects: MutableList<ObjectInInventarizedFile> = arrayListOf()

) {

companion object {

**private** const **val** BEGIN\_TABLE\_VALUE = "номерпопорядку"

}

**private** lateinit **var** savedSheet: Sheet

**private** lateinit **var** savedWorkbook: Workbook

**private** lateinit **var** uri: Uri

@Throws(

FileNotValidException::class, IOException::class, EncryptedDocumentException::**class**

)

**fun** **init**(inputStream: InputStream, uri: Uri): Job {

**this**.uri = uri

**return** **CoroutineScope**(Dispatchers.IO).launch {

**val** workbook = WorkbookFactory.create(inputStream)

**val** workSheet = workbook.getSheetAt(**0**)

savedSheet = workSheet

savedWorkbook = workbook

initTableColumnsPositions(workSheet)

setListOfObjects(workSheet)

}

}

**fun** **flushFactInventarizedDataInExcel**(outputStream: OutputStream) {

**for** (invObject **in** listOfObjects) {

invObject.writeFactDataInExcel(savedSheet, outputStream)

}

savedWorkbook.write(outputStream)

savedWorkbook.close()

}

**fun** **getUri**(): Uri {

**return** uri

}

//<Ищем стрелочку цифру рядом со стрелкой, чтобы потом определить позиции колонок>

//Номер по порядку

//

//->//1(номер кол) 2 (название) 3 ...

//1 Сервер-5

//2 Сервер-6

//3 Пистолет

//4 Воздушный шар

**private** **fun** **getFirstTableColumnExcelCell**(workSheet: Sheet): ExcelCellInfo {

**val** rowIterator = workSheet.rowIterator()

**for** (column **in** **0.**.**5**) {

**var** rowNumber = **0**

**while** (rowIterator.hasNext()) {

**var** cellValue = rowIterator.next().getCell(column)

**if** (cellValue != **null**) {

**val** checkingValue = cellValue.toString().replace(Regex("(\\s|-)\*"), "")

**if** (checkingValue.lowercase().contentEquals(BEGIN\_TABLE\_VALUE)) {

**for** (k **in** **1.**.**15**) {

cellValue = workSheet.getRow(rowNumber + k).getCell(column)

**if** (cellValue.cellType == CellType.NUMERIC) {

**if** (cellValue.numericCellValue.toInt() == **1**) {

**return** **ExcelCellInfo**(rowNumber + k, column)

}

} **else** {

**if** (cellValue.toString().contentEquals("1")) {

**return** **ExcelCellInfo**(rowNumber + k, column)

}

}

}

**return** **ExcelCellInfo**()

}

}

rowNumber++

}

}

**return** **ExcelCellInfo**()

}

**private** **fun** **addNewDocumentColumn**(

currDocColumnCounter: Int,

rowExcelPos: Int,

columnExcelPos: Int

) {

listOfDocumentColumns[currDocColumnCounter] = InventarizedColumn(

inventarizedColumnName = mapWithColumnNumberToNameBijection.getOrDefault(

currDocColumnCounter + **1**,

InventarizedColumnNames.UNKNOWN

),

excelCellInfo = ExcelCellInfo(row = rowExcelPos, column = columnExcelPos)

)

}

@Throws(FileNotValidException::class)

**private** **fun** **initTableColumnsPositions**(workSheet: Sheet) {

**val** startCellOfTable = getFirstTableColumnExcelCell(workSheet)

**if** (startCellOfTable.row == ExcelCellInfo.ERROR\_POS || startCellOfTable.column == ExcelCellInfo.ERROR\_POS) {

**throw** **FileNotValidException**("Begin of table not found.")

}

**var** lastCheckedColumn = startCellOfTable.column

**var** currDocColumn = **0**

**val** cellIterator = workSheet.getRow(startCellOfTable.row).cellIterator()

**while** (cellIterator.hasNext() && currDocColumn != valueOfTableColumnsInDocument) {

**val** checkingCell = cellIterator.next()

**if** (checkingCell.cellType == CellType.NUMERIC) {

**if** (checkingCell.numericCellValue.toInt() == currDocColumn + **1**) {

addNewDocumentColumn(currDocColumn, startCellOfTable.row, lastCheckedColumn)

currDocColumn++

}

} **else** {

**if** (checkingCell.toString().contentEquals((currDocColumn + **1**).toString())) {

addNewDocumentColumn(currDocColumn, startCellOfTable.row, lastCheckedColumn)

currDocColumn++

}

}

lastCheckedColumn++

}

**if** (currDocColumn != valueOfTableColumnsInDocument) {

**throw** **FileNotValidException**("Can not find column in document table. It is too far from previous column")

}

}

**private** **fun** **getCellFromRowByNeededColumn**(

row: Row,

inventarizedColumnName: InventarizedColumnNames

): Cell? {

**return** row.getCell(listOfDocumentColumns[inventarizedColumnName.columnNumber - **1**].excelCellInfo.column)

}

**private** **fun** **insertNewInventarizedObjectInListByRow**(row: Row, rowIndex: Int) {

**val** cellForOrdNumber = getCellFromRowByNeededColumn(

row,

InventarizedColumnNames.ORD\_NUMBER

)!!

**val** cellForReleaseYear = getCellFromRowByNeededColumn(

row,

InventarizedColumnNames.RELEASE\_YEAR

)!!

**val** cellForInvNumber =

getCellFromRowByNeededColumn(row, InventarizedColumnNames.INV\_NUMBER)!!

**val** cellForFactoryNumber =

getCellFromRowByNeededColumn(row, InventarizedColumnNames.FACTORY\_NUMBER)!!

**val** cellForPassportNumber =

getCellFromRowByNeededColumn(row, InventarizedColumnNames.FACTORY\_NUMBER)!!

**val** cellForAccountantQuantity =

getCellFromRowByNeededColumn(row, InventarizedColumnNames.ACCOUNTANT\_QUANTITY)!!

**val** cellForAccountantPrice =

getCellFromRowByNeededColumn(row, InventarizedColumnNames.ACCOUNTANT\_PRICE)!!

**val** cellForFactQuantity =

getCellFromRowByNeededColumn(row, InventarizedColumnNames.FACT\_QUANTITY)

**val** cellForFactPrice =

getCellFromRowByNeededColumn(row, InventarizedColumnNames.FACT\_PRICE)

listOfObjects.add(

ObjectInInventarizedFile(

ordNumber = **if** (cellForOrdNumber.cellType == CellType.NUMERIC) cellForOrdNumber.numericCellValue.toLong() **else** cellForOrdNumber.stringCellValue.toLong(),

objectName = getCellFromRowByNeededColumn(

row,

InventarizedColumnNames.OBJ\_NAME

)!!.stringCellValue,

docRentNumber = getCellFromRowByNeededColumn(

row,

InventarizedColumnNames.DOC\_RENT\_NUMBER

)!!.stringCellValue,

docRentName = getCellFromRowByNeededColumn(

row,

InventarizedColumnNames.DOC\_RENT\_NAME

)!!.stringCellValue,

releaseYear = **if** (cellForReleaseYear.cellType == CellType.NUMERIC) cellForReleaseYear.numericCellValue.toInt()

**else** **if** (cellForReleaseYear.stringCellValue.isEmpty()) **0** **else** cellForReleaseYear.stringCellValue.toInt(),

invNumber = **if** (cellForInvNumber.cellType == CellType.NUMERIC) cellForInvNumber.numericCellValue.toString() **else** cellForInvNumber.stringCellValue,

factoryNumber = **if** (cellForFactoryNumber.cellType == CellType.NUMERIC) cellForFactoryNumber.numericCellValue.toString() **else** cellForFactoryNumber.stringCellValue,

passportNumber = **if** (cellForPassportNumber.cellType == CellType.NUMERIC) cellForPassportNumber.numericCellValue.toString() **else** cellForPassportNumber.stringCellValue,

factQuantityAndPosInExcel = FieldAndExcelPosition(

cellForFactQuantity?.let {

**if** (cellForFactQuantity.cellType == CellType.NUMERIC)

cellForFactQuantity.numericCellValue.toInt()

**else** **if** (cellForFactQuantity.stringCellValue.isEmpty()) **0** **else** cellForFactQuantity.stringCellValue.toInt()

}

?: **0**,

ExcelCellInfo(

rowIndex,

listOfDocumentColumns[InventarizedColumnNames.FACT\_QUANTITY.columnNumber - **1**].excelCellInfo.column

)

),

factPriceAndPosInExcel = FieldAndExcelPosition(

cellForFactPrice?.let {

**if** (cellForFactPrice.cellType == CellType.NUMERIC)

BigDecimal(cellForFactPrice.numericCellValue)

**else** **if** (cellForFactPrice.stringCellValue.isEmpty()) BigDecimal.ZERO **else** BigDecimal(

cellForFactPrice.stringCellValue

)

} ?: BigDecimal.ZERO,

ExcelCellInfo(

rowIndex,

listOfDocumentColumns[InventarizedColumnNames.FACT\_PRICE.columnNumber - **1**].excelCellInfo.column

)

),

accountantQuantity =

**if** (cellForAccountantQuantity.cellType == CellType.NUMERIC)

cellForAccountantQuantity.numericCellValue.toInt()

**else** **if** (cellForAccountantQuantity.stringCellValue.isEmpty()) **0** **else** cellForAccountantQuantity.stringCellValue.toInt(),

accountantPrice = BigDecimal(

**if** (cellForAccountantPrice.cellType == CellType.NUMERIC) cellForAccountantPrice.numericCellValue.toString()

**else** **if** (cellForAccountantPrice.stringCellValue.isEmpty()) "0" **else** cellForAccountantPrice.stringCellValue

),

)

)

}

**private** **fun** **setListOfObjects**(workSheet: Sheet) {

**val** rowIterator = workSheet.rowIterator()

**var** currentRow = **0**

**while** (rowIterator.hasNext() && currentRow != listOfDocumentColumns[**0**].excelCellInfo.row + **1**) {///доходим до начала таблицы

rowIterator.next()

currentRow++

}

**while** (rowIterator.hasNext()) { //Ходим по строкам

**val** row = rowIterator.next()

**val** cellValue = getCellFromRowByNeededColumn(row, InventarizedColumnNames.ORD\_NUMBER)

?: **break**

**if** (!"(\\d)+(\\.)?(\\d)\*".toRegex().matches(cellValue.toString())) {

**break**

}

insertNewInventarizedObjectInListByRow(row, currentRow)

currentRow++

}

}

}

**enum** **class** **InventarizedColumnNames**(**val** columnNumber : Int) {

UNKNOWN(**0**),

ORD\_NUMBER(**1**), //порядковый номер

OBJ\_NAME(**2**), // наименование объекта

DOC\_RENT\_NAME(**3**), //имя документа арендованного объекта

DOC\_RENT\_DATE(**4**), //дата документа аренды объекта

DOC\_RENT\_NUMBER(**5**), // номер документа аренды объекта

RELEASE\_YEAR(**6**), // дата выпуска

INV\_NUMBER(**7**), // инвентарный номер

FACTORY\_NUMBER(**8**), // заводской номер

PASSPORT\_NUMBER(**9**), // номер паспорта (документа о регистрации)

FACT\_QUANTITY(**10**), //фактическое количество

FACT\_PRICE(**11**), // фактическая стоимость

ACCOUNTANT\_QUANTITY(**12**), // количество по данным бух. учета

ACCOUNTANT\_PRICE(**13**); // стоимость по данным бух. учета

//возвращает номер колонки по документу (не номер колонки в таблице)

}

data **class** **InventarizedColumn**(

**val** inventarizedColumnName: InventarizedColumnNames = InventarizedColumnNames.UNKNOWN,

**val** excelCellInfo: ExcelCellInfo = ExcelCellInfo()

)

data **class** **FieldAndExcelPosition**<T>(**var** fieldValue: T, **val** excelPos: ExcelCellInfo)

data **class** **ExcelCellInfo**(

**val** row: Int = ERROR\_POS,

**val** column: Int = ERROR\_POS

){

companion object{

const **val** ERROR\_POS = -**1**

}

}

data **class** **ObjectInInventarizedFile**(

**var** ordNumber: Long = **0L**,

**var** objectName: String = "",

**var** docRentName: String = "",

**var** docRentDate: Date = Date(),

**var** docRentNumber: String = "",

**var** releaseYear: Int = **0**,

**var** invNumber: String = "",

**var** factoryNumber: String = "",

**var** passportNumber: String = "",

**var** factQuantityAndPosInExcel: FieldAndExcelPosition<Int> = FieldAndExcelPosition(

**0**,

ExcelCellInfo()

),

**var** factPriceAndPosInExcel: FieldAndExcelPosition<BigDecimal> = FieldAndExcelPosition(

BigDecimal.ZERO,

ExcelCellInfo()

),

**var** accountantQuantity: Int = **0**,

**var** accountantPrice: BigDecimal = BigDecimal.ZERO,

**var** pricePerOne: BigDecimal = BigDecimal.ZERO

) {

init {

**if** (accountantQuantity != **0**) {

pricePerOne = accountantPrice / accountantQuantity.toBigDecimal()

}

}

**fun** **incrementFactQuantity**() {

factQuantityAndPosInExcel.fieldValue++

factPriceAndPosInExcel.fieldValue =

pricePerOne.multiply(BigDecimal(factQuantityAndPosInExcel.fieldValue))

}

**fun** **writeFactDataInExcel**(workSheet: Sheet, outputStream: OutputStream) {

**val** factQuantityExcelPost = factQuantityAndPosInExcel.excelPos

**val** factPriceExcelPost = factPriceAndPosInExcel.excelPos

workSheet.getRow(factQuantityExcelPost.row).getCell(factQuantityExcelPost.column)

.setCellValue(factQuantityAndPosInExcel.fieldValue.toDouble())

workSheet.getRow(factPriceExcelPost.row).getCell(factPriceExcelPost.column)

.setCellValue(factPriceAndPosInExcel.fieldValue.toDouble())

}

}

**enum** **class** **LoadStatus** {

SUCCESS, ERROR\_OPENING\_FILE, ERROR\_PARSING\_FILE, LOADING, NO\_FILE, UNKNOWN\_ERROR

}

**class** **FileNotValidException**(messageInfo: String) : Exception(messageInfo)

## ПРИЛОЖЕНИЕ Э

|  |  |
| --- | --- |
| Рисунок 56 – Перед проведение инвентаризации | Рисунок 57 – Сверка по инвентарному номеру |

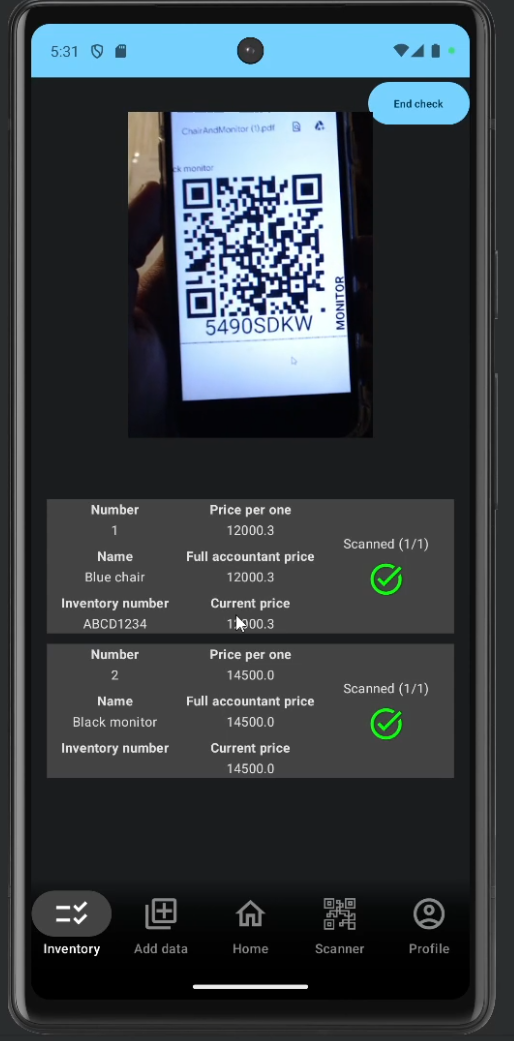


Рисунок 58 – Сверка по имени при помощи сервера

## ПРИЛОЖЕНИЕ Ю

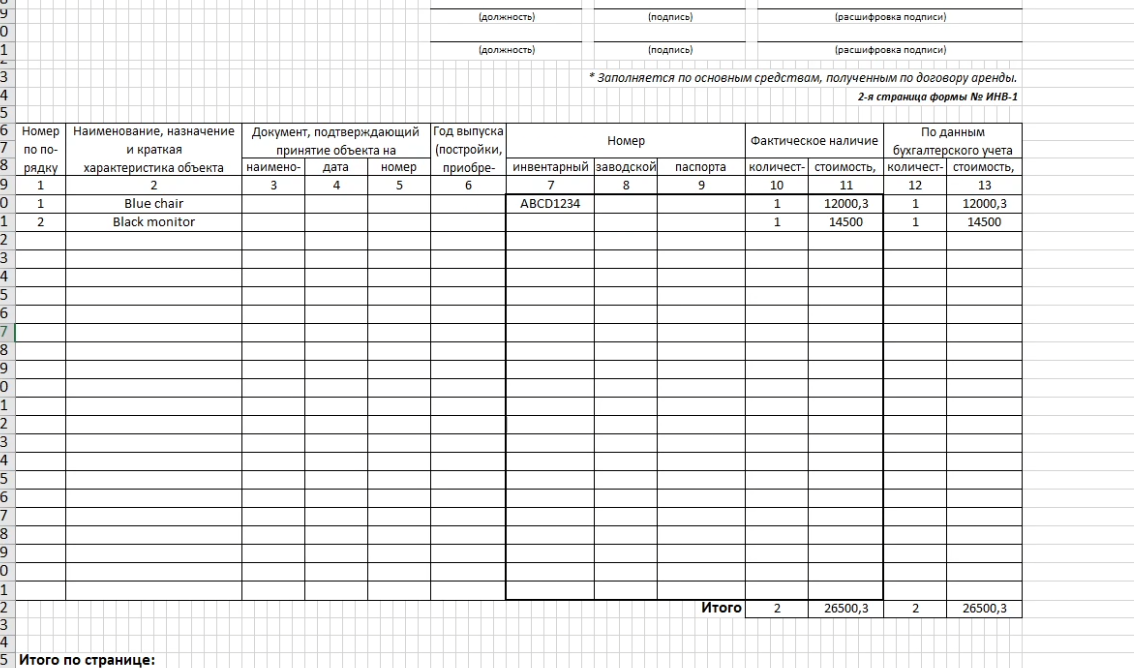


Рисунок 59 – Результирующая таблица после инвентаризации

## ПРИЛОЖЕНИЕ Я

|  |  |
| --- | --- |
| Рисунок 60 – Ссылка на клиентскую часть | Рисунок 61 – Ссылка на серверную часть |