МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ   
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

**«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ**

**ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ им. В. Г. ШУХОВА»**

**(БГТУ им. В.Г. Шухова)**

Кафедра программного обеспечения вычислительной техники и

автоматизированных систем

**КУРСОВАЯ РАБОТА**

по дисциплине: **Базы данных**

по теме: “Мобильный справочник для настольной ролевой игры”

Автор работы\_\_\_\_\_\_\_\_\_\_\_Дмитриев Андрей Александрович ПВ-223

Руководитель проекта\_\_\_\_\_\_\_\_\_\_\_ Панченко Максим Владимирович

Оценка\_\_\_\_\_\_

Белгород 2024

**Содержание**

[Введение 3](#_Toc186028897)

[1. Выбор архитектуры приложения и схемы базы данных 4](#_Toc186028898)

[1.1. Архитектура приложения 4](#_Toc186028899)

[1.2. Схема базы данных 5](#_Toc186028900)

[2. Варианты запросов для реализации фильтрации и поиска 7](#_Toc186028901)

[2.1. Запрос локализованных объектов 7](#_Toc186028902)

[2.2. Запрос объектов из пользовательского сборника 9](#_Toc186028903)

[3. Реализация разграничения доступа и функционал ролей 10](#_Toc186028904)

[3.1. Разграничение доступа 10](#_Toc186028905)

[3.2. Реализация функционала ролей 10](#_Toc186028906)

[4. Экспорт данных в избранные форматы 11](#_Toc186028907)

[4.1. Экспорт в JSON 11](#_Toc186028908)

[4.2. Экспорт в PDF 12](#_Toc186028909)

[5. Создание дампа базы данных 13](#_Toc186028910)

[Заключение 15](#_Toc186028911)

[Список источников и литературы 16](#_Toc186028912)

[Приложения 17](#_Toc186028913)

# Введение

Курсовой проект представляет собой статический справочник заклинаний для настольной ролевой игры. СУБД выбрана SQLite, так как данные хранятся исключительно в памяти устройства. ORM выбрана Room – библиотека от Google. Для разработки интерфейса и логики использована библиотека Compose.

Целью данной работы является создание мобильного приложения-справочника с поддержкой фильтрации и поиска по нему, а также поддержкой нескольких языков и прочего функционала, определённого заданием к курсовой работе.

Для достижения поставленной цели необходимо решить следующие задачи:

1. Выбор архитектуры приложения и схемы базы данных.
2. Варианты запросов для реализации фильтрации и поиска
3. Реализация разграничения доступа и функционал ролей
4. Экспорт данных в избранные форматы
5. Создание дампа базы данных

# 1. Выбор архитектуры приложения и схемы базы данных

## 1.1. Архитектура приложения

Была избрана «чистая архитектура». Она декларирует разделение на 3 модуля: data, domain и app (Рисунок 1).

Data содержит функционал для работы с удалённым или встроенным хранилищем. Здесь в проекте организована работа с локальной базой данных и Preferences (специальное хранилище «ключ-значение»).

Domain содержит исключительно бизнес логику, в том числе связанную с использованием хранилища.

App (presentation) применяет бизнес логику и напрямую связан с мобильным устройством. В ней прописываются разрешения, ресурсы (строки, изображения, стили), макеты, обработчики состояний, «инъекции» зависимостей и т.д.

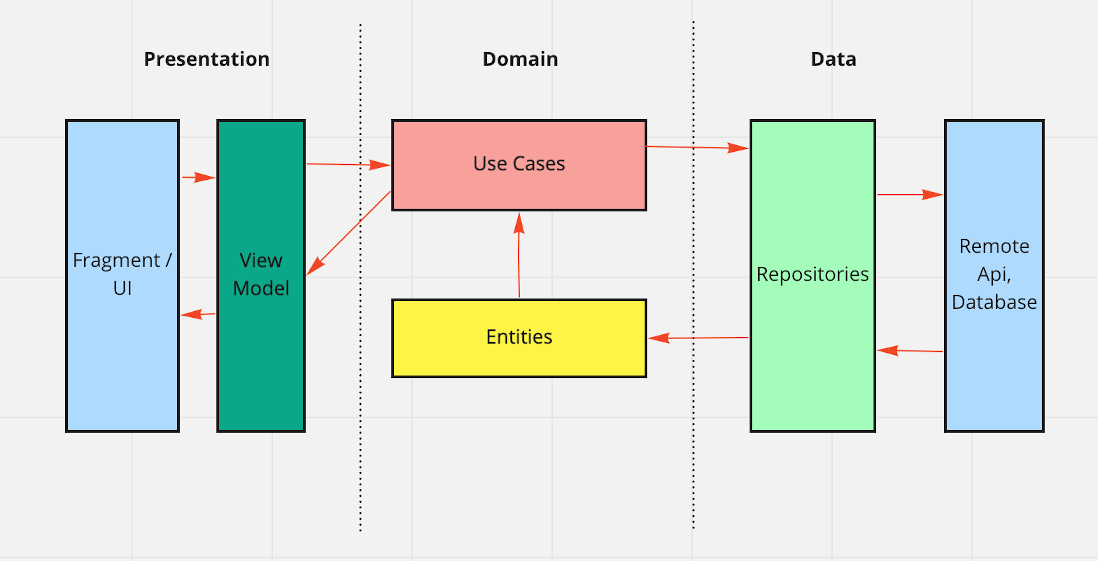


Рисунок 1. Чистая архитектура

## 1.2. Схема базы данных

В приложении будет локализовываться в зависимости от языка для этого требуется предпринять некоторые меры (Рисунок 2).



Рисунок 2. Схема базы данных.

Таблица tagging\_spell представляет набор тегов, которые соответственно не зависит от локализации, они имеет отношение один к одному с таблицей spell, которая содержит локализованные json.

Таблица book будет содержать пользовательские «книги заклинаний».

Описание схемы в синтаксисе ORM представлено в листинге 1.

@Entity(tableName = BookEntity.TABLE\_NAME)  
data class BookEntity(  
 @PrimaryKey(autoGenerate = true)  
 @ColumnInfo(name = COLUMN\_ID)  
 val id: Long = 0,  
 @ColumnInfo(name = COLUMN\_NAME)  
 val name: String  
) {  
 companion object {  
 const val TABLE\_NAME = "book"  
 const val COLUMN\_ID = "id"  
 const val COLUMN\_NAME = "name"  
 }  
}

@Entity(  
 tableName = BooksSpellsXRefEntity.TABLE\_NAME,  
 primaryKeys = [  
 BooksSpellsXRefEntity.COLUMN\_BOOK\_ID,  
 BooksSpellsXRefEntity.COLUMN\_SPELL\_UUID  
 ]  
)  
class BooksSpellsXRefEntity(  
 @ColumnInfo(name = COLUMN\_BOOK\_ID)  
 val bookId: Long,  
 @ColumnInfo(name = COLUMN\_SPELL\_UUID)  
 val spellUuid: String  
) {  
 companion object {  
 const val TABLE\_NAME = "books\_with\_spells"  
 const val COLUMN\_BOOK\_ID = "book\_id"  
 const val COLUMN\_SPELL\_UUID = "spell\_uuid"  
 }  
}

@Entity(tableName = TaggingSpellEntity.TABLE\_NAME)  
data class TaggingSpellEntity(  
 @PrimaryKey  
 @ColumnInfo(name = COLUMN\_UUID)  
 val uuid: String,  
 @ColumnInfo(name = COLUMN\_LEVEL\_TAG)  
 val levelTag: String? = null,  
 @ColumnInfo(name = COLUMN\_SCHOOL\_TAG)  
 val schoolTag: String? = null,  
 @ColumnInfo(name = COLUMN\_CASTING\_TIME\_TAG)  
 val castingTime: String? = null,  
 @ColumnInfo(name = COLUMN\_RANGE\_TAG)  
 val rangeTag: String? = null,  
 @ColumnInfo(name = COLUMN\_RITUAL\_TAG)  
 val ritualTag: String? = null,  
 @ColumnInfo(name = COLUMN\_SOURCE\_TAG)  
 val sourceTag: String? = null  
  
 // *todo add more columns here*) {  
 companion object {  
 const val TABLE\_NAME = "tagging\_spell"  
 const val COLUMN\_UUID = "uuid"  
 const val COLUMN\_SCHOOL\_TAG = "school\_tag"  
 const val COLUMN\_LEVEL\_TAG = "level\_tag"  
 const val COLUMN\_CASTING\_TIME\_TAG = "casting\_time"  
 const val COLUMN\_RANGE\_TAG = "range\_tag"  
 const val COLUMN\_RITUAL\_TAG = "ritual\_tag"  
 const val COLUMN\_SOURCE\_TAG = "source\_tag"  
  
 // *todo add more names for columns here* }  
}

@Entity(  
 tableName = SpellEntity.TABLE\_NAME,  
 primaryKeys = [*COLUMN\_UUID*, *COLUMN\_LANGUAGE*],  
 foreignKeys = [  
 ForeignKey(  
 entity = TaggingSpellEntity::class,  
 parentColumns = [TaggingSpellEntity.COLUMN\_UUID],  
 childColumns = [*COLUMN\_UUID*]  
 )  
 ]  
)  
class SpellEntity(  
 @ColumnInfo(name = COLUMN\_UUID)  
 val uuid: String,  
 @ColumnInfo(name = COLUMN\_LANGUAGE)  
 val language: String = LocaleEnum.*DEFAULT*.value,  
 @ColumnInfo(name = COLUMN\_NAME)  
 val name: String,  
 @ColumnInfo(name = COLUMN\_JSON)  
 val json: String  
) {  
 companion object {  
 const val TABLE\_NAME = "spell"  
 const val COLUMN\_UUID = "uuid"  
 const val COLUMN\_LANGUAGE= "language"  
 const val COLUMN\_NAME = "name"  
 const val COLUMN\_JSON = "json"  
 }  
}

Листинг 1. Описание в синтаксисе ORM

# 2. Варианты запросов для реализации фильтрации и поиска

## 2.1. Запрос локализованных объектов

Для запроса списка заклинаний используется запрос из листинга 2. Этот запрос конструируется из двух частей: getSpellsWithTagsShortQuery – селект запрос локализованных заклинаний и filterSuffixQuery – условие where для фильтрации и сортировки.

suspend fun getSpellsShort(  
 filter: Map<TagIdentifierEnum, List<TagEnum>> = *emptyMap*(),  
 sorter: SortOptionEnum = SortOptionEnum.*BY\_NAME*,  
 language: LocaleEnum = LocaleEnum.*ENGLISH*): List<SpellWithTagsShort> =  
 getManyShort(  
 SimpleSQLiteQuery(  
 *getSpellsWithTagsShortQuery*(language)  
 + *filterSuffixQuery*(filter, sorter)  
 )  
 )

internal fun getSpellsWithTagsShortQuery(  
 language: LocaleEnum  
) = "select \* from ${TaggingSpellEntity.TABLE\_NAME} as t0 " +  
 "inner join ${SpellEntity.TABLE\_NAME} as t1 " +  
 "on t0.${SpellEntity.COLUMN\_UUID}=t1.${TaggingSpellEntity.COLUMN\_UUID} " +  
 "and t1.${SpellEntity.COLUMN\_LANGUAGE} in ('${language.value}', '${LocaleEnum.*DEFAULT*.value}')"  
  
internal fun filterSuffixQuery(  
 filter: Map<TagIdentifierEnum, List<TagEnum>> = *emptyMap*(),  
 sorter: SortOptionEnum = SortOptionEnum.*BY\_NAME*) = StringBuilder().*apply* **{** // begin condition  
 append(" where 1=1 ")  
 // set filters  
 filter.*forEach* **{** entry **->** if (entry.value.*isNotEmpty*())  
 append("and ${entry.key.*toColumnName*()} in (${entry.value.*toTableFields*()}) ")  
 **}** // set sorter  
 when (sorter) { SortOptionEnum.*BY\_NAME* -> Unit  
  
 SortOptionEnum.*BY\_LEVEL* ->  
 append(", ${TaggingSpellEntity.COLUMN\_LEVEL\_TAG} asc ")  
  
 else -> throw IllegalArgumentException("sort option not supported")  
 }  
 append("order by ${SpellEntity.COLUMN\_NAME} asc")  
**}**.toString()

Листинг 2. Запрос локализованных данных.

Вывод данного запроса представлен на рисунке 3 и представляет список заклинаний, а именно на русском, на английском и также отфильтрованный по тегам. Теги описывают большую часть характеристик заклинаний. Элемент списка содержит «название» и «уровень», «название» получено из таблицы spell, а «уровень» из tagging\_spell.

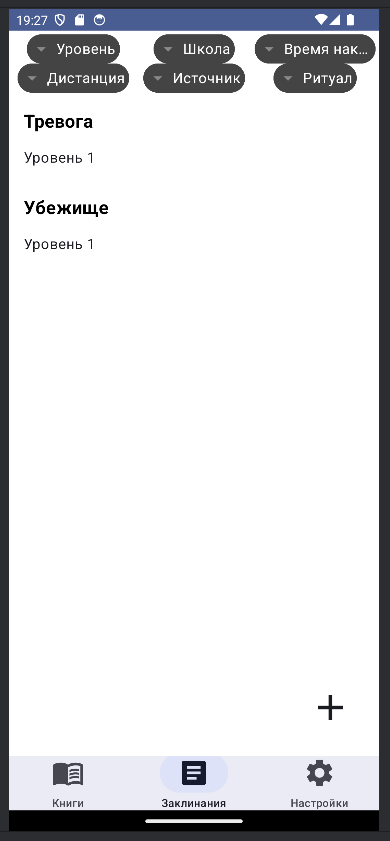
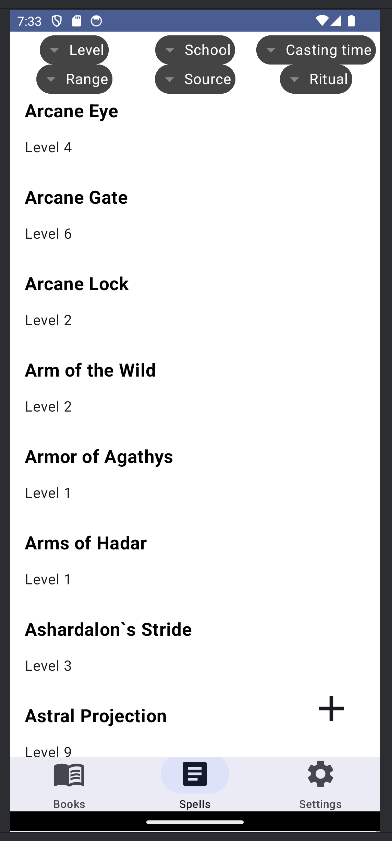
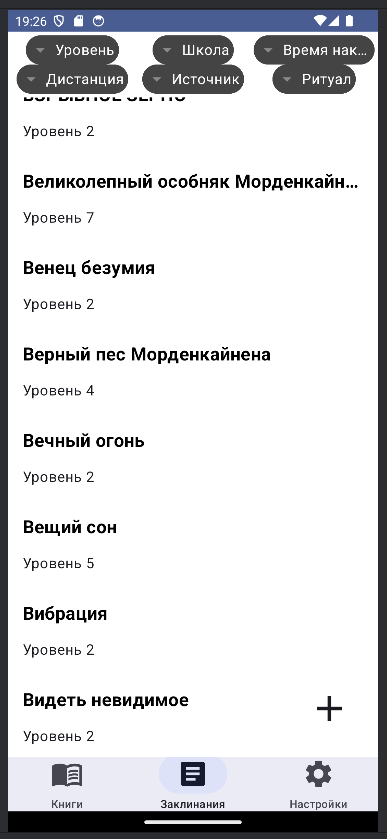


Рисунок 3. Вывод всех заклинаний.

Если нажать на элемент списка, то откроется подробная информация о нём (Листинг 3, Рисунок 4). Здесь данные получаемые исключительно из таблицы spell.

@Query(  
 "select \* from ${SpellEntity.TABLE\_NAME} " +  
 "where ${SpellEntity.COLUMN\_UUID} = :uuid " +  
 "and ${SpellEntity.COLUMN\_LANGUAGE} in (:language, 'default') " +  
 "limit 1"  
)  
abstract suspend fun getSpellDetail(uuid: String, language: String): SpellEntity

Листинг 3. Запрос подробной информации.

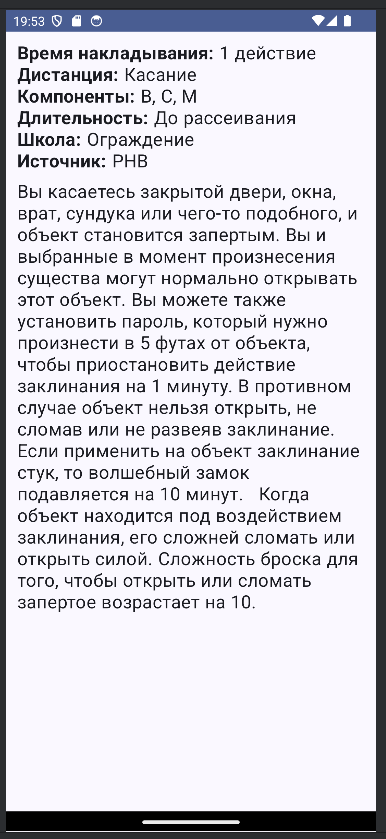


Рисунок 4. Вывод подробной информации

## 2.2. Запрос объектов из пользовательского сборника

Пользователь может собрать книгу, которая будет содержать выбираемые заклинания (Рисунок 5).

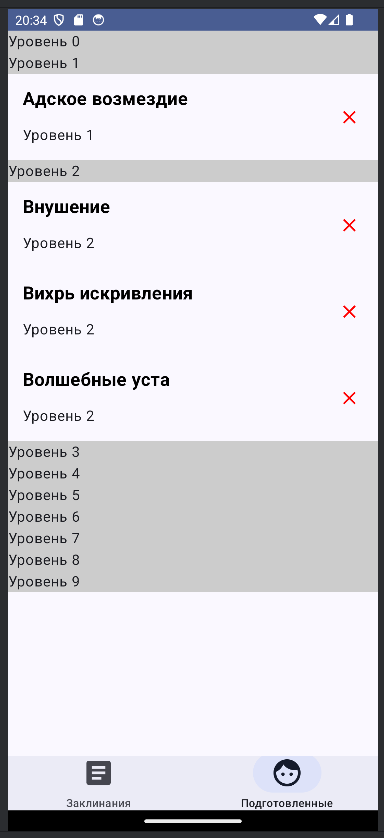
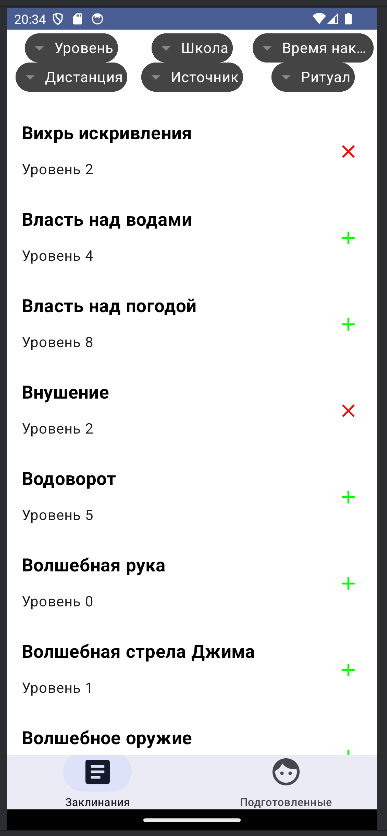
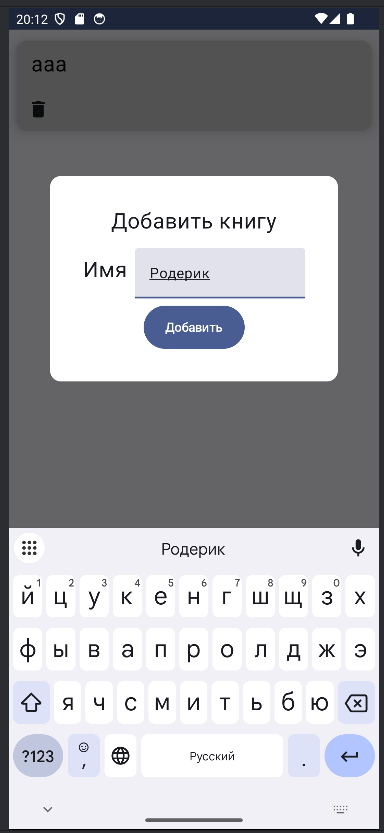


Рисунок 5. Пользовательский сборник.

Заклинания из книги во вкладке «подготовленные» выбираются с сортировкой по названию, и вскоре упорядочиваются по «уровням» с помощью связной хэш-таблицы.

# 3. Реализация разграничения доступа и функционал ролей

## 3.1. Разграничение доступа

Разграничение доступа реализовано на слое приложения. Выделено 2 роли: обычный пользователь и платный пользователь. Различия у них в том, что платный пользователь может создавать свои заклинания (Рисунок 6).

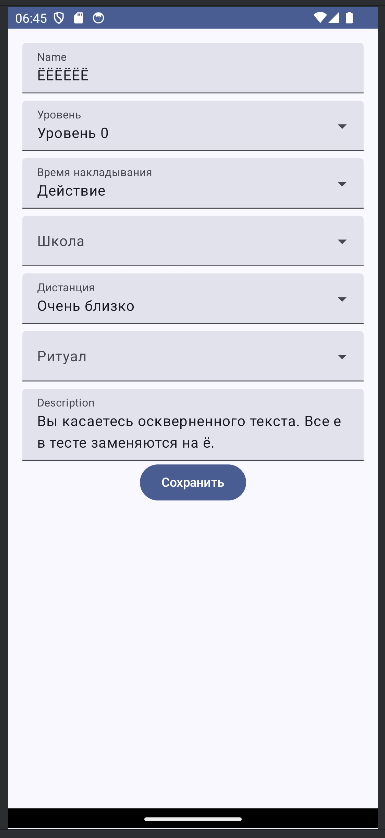
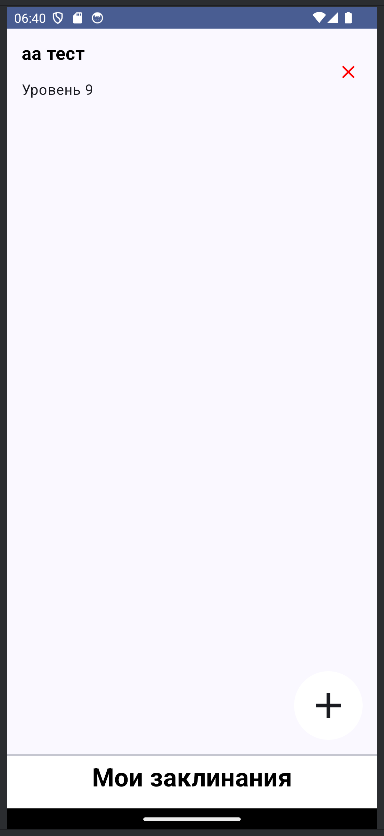
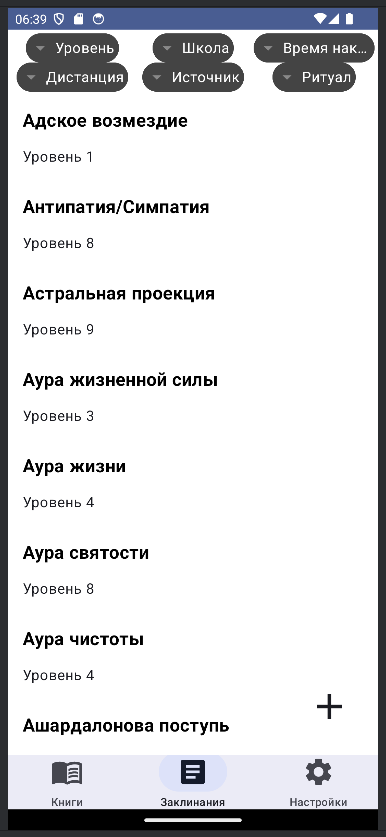
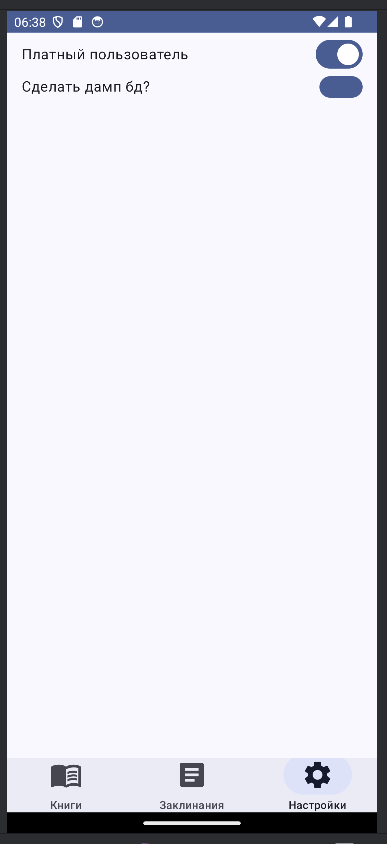


Рисунок 6. Интерфейс платного пользователя.

## 3.2. Реализация функционала ролей

В системе Android есть специальное хранилище – Preferences – предназначенное для сохранения примитивных данных по ключу, обычно оно используется для настроек. В этом хранилище и сохраняется роль пользователя.

В коде созданы usecase-ы, которыми можно управлять и читать настройки. В зависимости от них появляются элементы интерфейса, которые не доступны обычному пользователю (Листинг 4).

Так как приложение мультиязычное, созданные заклинания не имеют языка и в столбце language пишется default. При всех запросах заклинаний этот параметр учитывается и новые заклинания отображаются.

floatingButton = **{** if (state.isPaidUser)  
 AddFloatingButton(  
 onClick = **{** navController.*navigate*(NavEndpoint.AuthorSpells)  
 **}** )  
**}**

Листинг 4. Логика отображения элементов интерфейса платного пользователя.

# 4. Экспорт данных в избранные форматы

## 4.1. Экспорт в JSON

Для выборки на экспорт используются заклинания, собранные в книгу.

Многие данные и так хранятся в JSON, поэтому выборка будет достаточно примитивна (Листинг 5). Usecase возвращает inputStream из которого формируется файл.

suspend fun execute(bookId: Long): Result<Pair<String, InputStream>> {  
 val book = bookRepository.get(bookId)  
 val spellsJson = spellRepository.getSpellsJsonByBook(bookId, localeEnum)  
  
 return withContext(Dispatchers.IO) **{** try {  
 Result.success(  
 Pair(  
 "book\_${book.name}.json",  
 JsonObject().*apply* **{** addProperty("name", book.name)  
 add("spells", JsonArray().*also* **{** arr **->** spellsJson.*forEach* **{**spellJson **->** arr.add(JsonParser.parseString(spellJson))  
 **}  
 }**)  
 **}** .toString()  
 .*byteInputStream*()  
 )  
 )  
 } catch (e: Exception) {  
 Result.failure(e)  
 }  
 **}**}

Листинг 5. Usecase запроса выборки.

Выборка выглядит следующим образом (Рисунок 7).

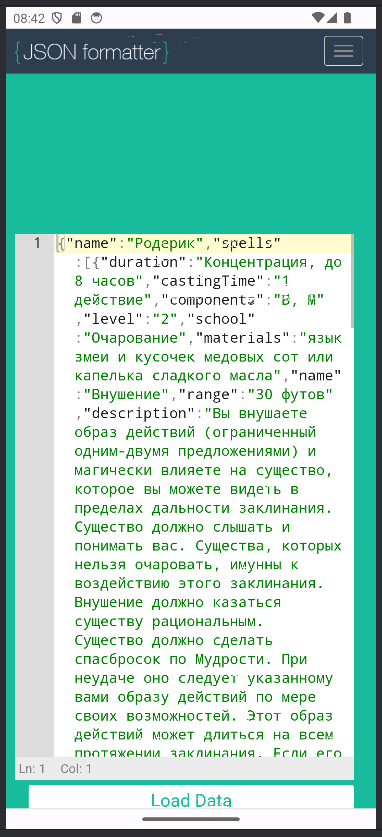


Рисунок 7. Пример JSON файла.

## 4.2. Экспорт в PDF

Выборка происходит с теми же параметрами. Большую сложность представляет отрисовка PDF (Рисунок 8). Для рисования выбираются только некоторые данные из выборки, чтобы сэкономить место в документе. Если текст не влезает, то он обрезается.

При формировании документа возникла проблема. Из-за архитектуры приложения сложно придумать способ перевести названия заголовков, поэтому они соответствуют ключам из JSON.

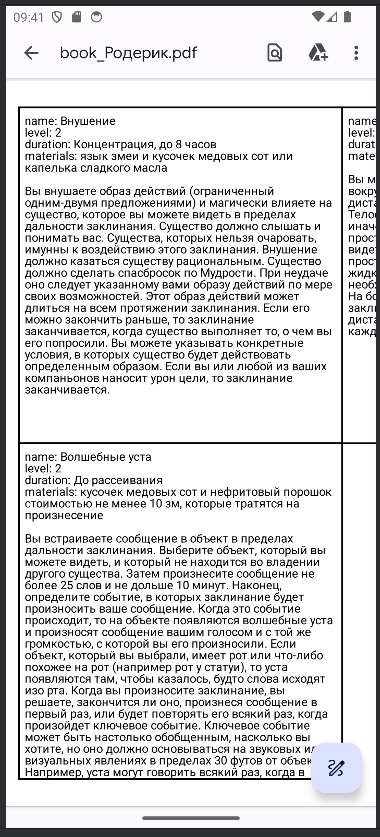


Рисунок 8. Пример PDF файла.

# 5. Создание дампа базы данных

Экспорт дампа можно произвести из настроек. База данных хранится в приложении по пути, получаемому функцией: context.getDatabasePath( AppDatabaseConnection.DB\_NAME) – и хранится по пути: /data/user/0/com.example.spellsbook/files/profileInstalled. Сохранение происходит в загрузки (Рисунок 9).

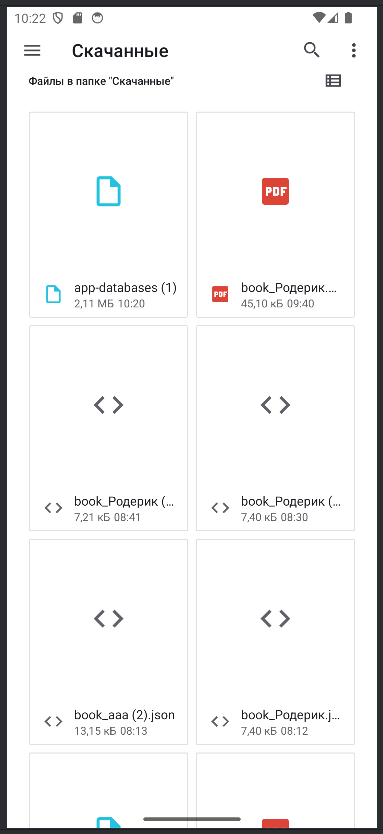
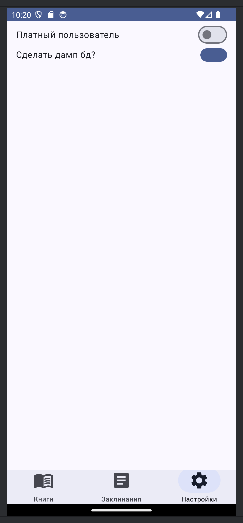


Рисунок 9. Экспорт дампа базы данных (app-databases (1)).

# Заключение

В ходе курсовой работы было разработано мобильное приложение для платформы Android с использованием СУБД SQLite и ORM Room. Реализована локализация средствами Android и базой данных. Создана логика составления файлов и их экспорт.

Расширены навыки владения инструментарием Android в связке с базами данных с применением популярных библиотек.

# Список источников и литературы

1. Develop for Android [Электронный ресурс] Режим доступа: <https://developer.android.com/develop>
2. Jetpack Compose UI App Development Toolkit [Электронный ресурс] Режим доступа: <https://developer.android.com/compose>
3. Save data in a local database using Room [Электронный ресурс] Режим доступа: <https://developer.android.com/training/data-storage/room>
4. Clean architecture в android для начинающих [Электронный ресурс] Режим доступа: <https://medium.com/nuances-of-programming/clean-architecture-%D0%B2-android-%D0%B4%D0%BB%D1%8F-%D0%BD%D0%B0%D1%87%D0%B8%D0%BD%D0%B0%D1%8E%D1%89%D0%B8%D1%85-f44d25495f5b>

# Приложения

Github проекта: <https://github.com/AnDreV133/SpellsBook2>