


Файлы и базы данных



Сохранение информации

Часто в приложении необходимо сохранить какую-либо информацию на долгое время. В современной Android разработке для этого существуют различные способы:

- **Файлы**, специфичные для приложения
- **Media** (Фото, видео, аудио)
- **Документы** и остальные файлы
- **Preferences** (ключ - значение)
- **Базы данных**

Файлы, специфичные для приложения

Т.е. файлы, которые могут использоваться только вашим приложением. Для манипулирования такими файлами не нужны специальные разрешения:

Files

Kotlin

```
context.filesDir  
context.cacheDir  
context.externalCacheDir  
context.getExternalFilesDir(Environment.DIRECTORY_DOCUMENTS)
```

Media

Видео, аудио и картинки, которыми может делиться ваше приложение.

В зависимости от версии Android, необходимы разрешения
READ_EXTERNAL_STORAGE,
WRITE_EXTERNAL_STORAGE,
READ_MEDIA_IMAGES и др.

Для работы с такими файлами
можно использовать:

- Photo picker API
- Media Store API (ContentResolver + SQL запросы)

Files

Kotlin

```
val projection = arrayOf(media-database-columns-to-retrieve)
val selection = sql-where-clause-with-placeholder-variables
val selectionArgs = values-of-placeholder-variables
val sortOrder = sql-order-by-clause

applicationContext.contentResolver.query(
    MediaStore.media-type.Media.EXTERNAL_CONTENT_URI,
    projection,
    selection,
    selectionArgs,
    sortOrder
)?.use { cursor ->
    while (cursor.moveToNext()) {
        // Use an ID column from the projection to get
        // a URI representing the media item itself.
    }
}
```

Документы

Другие типы документов, включая загруженные файлы. Для работы с ними **не нужны** специальные разрешения.

Необходимо использовать Storage Access Framework.

Files

Kotlin

```
fun openFile(pickerInitialUri: Uri) {  
    val intent = Intent(Intent.ACTION_OPEN_DOCUMENT).apply {  
        addCategory(Intent.CATEGORY_OPENABLE)  
        type = "application/pdf"  
  
        // Optionally, specify a URI for the file that should appear in the  
        // system file picker when it loads.  
        putExtra/DocumentsContract.EXTRA_INITIAL_URI, pickerInitialUri)  
    }  
  
    startActivityForResult(intent, PICK_PDF_FILE)  
}
```

Shared Preferences

Если необходимо хранить небольшое количество информации в формате ключ-значение, можно использовать **SharedPreferences API**, которые предоставляет удобные методы записи и чтения таких данных. Файл с данными будет храниться в **context.filesDir**.

Files

Kotlin

```
val sharedPref1 = activity?.getSharedPreferences(  
    getString(R.string.preference_file_key),  
    Context.MODE_PRIVATE  
)  
val sharedPref2 = activity?.getPreferences(Context.MODE_PRIVATE)
```

Files

Kotlin

```
with (sharedPref.edit()) {  
    putInt(getString(R.string.saved_high_score_key), newHighScore)  
    apply()  
}
```

Files

Kotlin

```
val defaultValue = resources.getInteger(R.integer.saved_high_score_default_key)  
val highScore = sharedPref.getInt(getString(R.string.saved_high_score_key), defaultValue)
```

Encrypted Shared Preferences

Однако, хранение файла с данными не может обеспечить полную защиту информации. Поэтому, если в ваших Shared Preferences лежат чувствительные данные, их нужно дополнительно шифровать. Ключ шифрования будет храниться в специальном системном хранилище.

Files

Java

```
val masterKeyAlias = MasterKeys.getOrCreate(MasterKeys.AES256_GCM_SPEC)

val sharedPreferences = EncryptedSharedPreferences.create(
    "PreferencesFilename",
    masterKeyAlias,
    applicationContext,
    EncryptedSharedPreferences.PrefKeyEncryptionScheme.AES256_SIV,
    EncryptedSharedPreferences.PrefValueEncryptionScheme.AES256_GCM
)
```



Datastore

Datastore - современный фреймворк, построенный на **flow** и **coroutines**, который заменяет стандартный SharedPreferences.

Существует 2 типа datastore:

- **Preferences Datastore** - как SharedPreferences сохраняет простые данные по ключу. Не обеспечивает безопасность типов.
- **Proto Datastore** - способен сохранять любые типы данных. Необходимо задавать схему хранения с помощью специального протокола, обеспечивает безопасность типов.

Preferences Datastor

- Получаем экземпляр нашего Preferences Datasore.
- В качестве ключа используются специальные объекты, создаваемые с помощью функции `{type}PreferenceKey(name)`.
- С помощью ключа получаем доступ к нашему полю из Preferences (Чтение и запись).

Files

Kotlin

```
val Context.dataStore: DataStore<Preferences> by preferencesDataStore(name = "settings")
```

Files

Kotlin

```
val EXAMPLE_COUNTER = intPreferencesKey("example_counter")
val exampleCounterFlow: Flow<Int> = context.dataStore.data
    .map { preferences ->
        // No type safety.
        preferences[EXAMPLE_COUNTER] ?: 0
    }
```

Files

Kotlin

```
suspend fun incrementCounter() {
    context.dataStore.edit { settings ->
        val currentCounterValue = settings[EXAMPLE_COUNTER] ?: 0
        settings[EXAMPLE_COUNTER] = currentCounterValue + 1
    }
}
```

Proto Datastore

Proto Datastore требует predeterminedенную схему, для хранения сложных типов. Схему необходимо хранить в директории **app/src/main/proto/**.

Схема пишется на языке **Protobuf**. Название message должно совпадать с классом, который мы будем хранить.

Files

Java

```
syntax = "proto3";

option java_package = "com.example.application";
option java_multiple_files = true;

message Settings {
    int32 example_counter = 1;
}
```

Proto Datastore

Далее необходимо создать специальный класс `Serializer<Type>`, который описывает, как необходимо записать в специальный `OutputStream` (поток байтов на запись) и как читать из `InputStream` (поток байтов на чтение).

При создании нашего `ProtoDatastore`, необходимо передать ему созданный `Serializer`.

Files

Kotlin

```
object SettingsSerializer : Serializer<Settings> {
    override val defaultValue: Settings = Settings.getDefaultInstance()

    override suspend fun readFrom(input: InputStream): Settings {
        try {
            return Settings.parseFrom(input)
        } catch (exception: InvalidProtocolBufferException) {
            throw CorruptionException("Cannot read proto.", exception)
        }
    }

    override suspend fun writeTo(
        t: Settings,
        output: OutputStream) = t.writeTo(output)
    }

    val Context.settingsDataStore: DataStore<Settings> by datastore(
        fileName = "settings.pb",
        serializer = SettingsSerializer
    )
}
```

Proto Datastore

- Для чтения данных используем проперти data для получения flow с нашими данными. Нам не нужно создавать ключи для доступа к данным, т.к. генерируются типобезопасные проперти из схемы.
- Для изменения данных вызываем метод updateData и используем паттерн строитель для получения новых данных.

Files

Kotlin

```
val exampleCounterFlow: Flow<Int> = context.settingsDataStore.data
    .map { settings ->
        // The exampleCounter property is generated from the proto schema.
        settings.exampleCounter
    }
```

Files

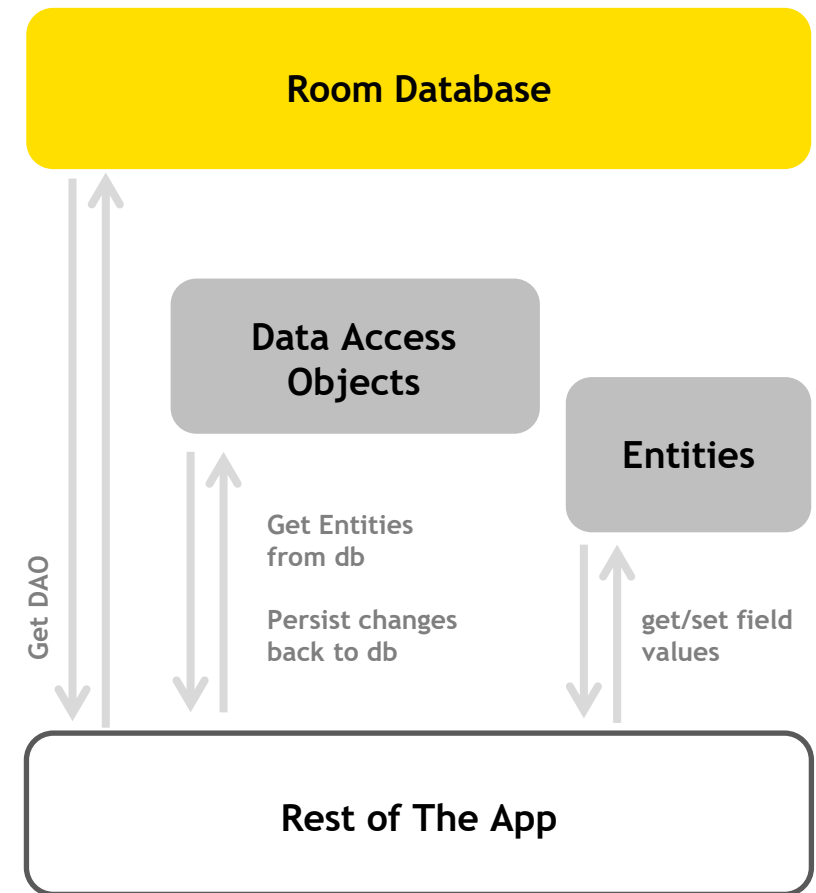
Kotlin

```
suspend fun incrementCounter() {
    context.settingsDataStore.updateData { currentSettings ->
        currentSettings.toBuilder()
            .setExampleCounter(currentSettings.exampleCounter + 1)
            .build()
    }
}
```

Базы данных

Однако для хранения большого количества данных, которые дополнительно могут быть связаны между собой, обычно принято использовать базы данных. На данный момент существуют различные реализации баз данных, например Realm или Room. В данной лекции поговорим именно про Room, как общепринятое решение.

Room является оберткой над **SQLite**. **SQLite** - быстрая, небольшая реализация SQL базы данных, разработанная специально для мобильных телефонов. **SQLite** поставляется разработчикам вместе с Android SDK.



Room Database

- Создаем классы-Entity, которые представляют таблицы в нашей бд. Здесь можно настроить первичные ключи, столбцы, значения по умолчанию и т.д.
- В одной бд может находиться много таблиц, поэтому и классов Entity может быть много.

Files

Kotlin

```
@Entity
data class User(
    @PrimaryKey val uid: Int,
    @ColumnInfo(name = "first_name") val firstName: String?,
    @ColumnInfo(name = "last_name") val lastName: String?
)
```

Room Database

- Далее нам необходимо определить DAO - Data Access Object, специальный интерфейс, который определяет, что мы можем делать с нашими данными.
- Существуют готовые операции, типа @Delete и @Insert, однако можно писать свои запросы на языке SQL с помощью @Query.
- Также по умолчанию функции могут быть suspend, а также можно возвращать Flow данных, если мы хотим отслеживать изменения.

Files

Kotlin

```
@Dao
interface UserDao {
    @Query("SELECT * FROM user")
    fun getAll(): List<User>

    @Query("SELECT * FROM user WHERE uid IN (:userIds)")
    fun loadAllByIds(userIds: IntArray): List<User>

    @Query("SELECT * FROM user WHERE first_name LIKE :first AND " +
        "last_name LIKE :last LIMIT 1")
    fun findByName(first: String, last: String): User

    @Insert
    fun insertAll(vararg users: User)

    @Delete
    fun delete(user: User)
}
```

Room Database

- В конце нам необходимо создать класс с базой данных, где мы определим, какие Entity в ней хранятся и каким DAO мы будем пользоваться.
- Остается только получить DAO из нашей базы данных и начать выполнять запросы.

Files

Kotlin

```
@Database(entities = [User::class], version = 1)
abstract class AppDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao
}
```

Files

Kotlin

```
val db = Room.databaseBuilder(
    applicationContext,
    AppDatabase::class.java, "database-name"
).build()
```

Files

Kotlin

```
val userDao = db.userDao()
val users: List<User> = userDao.getAll()
```


SQL CIPHER

- Базы данных также не защищены полностью от чужих атак, поэтому есть способы зашифровать и их.
- Для Room, например, можно использовать SQLCipher библиотеку <https://github.com/sqlcipher/android-database-sqlcipher>

Files

Java

```
final byte[] passphrase = SQLiteDatabase.getBytes(userEnteredPassphrase);
final SupportFactory factory = new SupportFactory(passphrase);
final SomeDatabase room = Room.databaseBuilder(activity, SomeDatabase.class, DB_NAME)
    .openHelperFactory(factory)
    .build();
```



True Engineering

630128, г. Новосибирск,
ул. Кутателадзе, 4г

(383) 363-33-51, 363-33-50
info@trueengineering.ru
trueengineering.ru

Новосибирский
Государственный
Университет