Хранение данных Content Provider

Клещин Никита

Содержание занятия

1. Для чего это надо?
2. savedInstanceState, но …
3. SharedPreferences и …
4. SQLite, и …
5. AccountManager, но …
6. но File Storage
7. ContentProvider

Для чего это надо?

• • • • • •

Поиграем в БИНГО

БИНГО

Кэширование
Сохранение каких-то данных, чтобы не повторять запросы

Хранение данных
Персистентное хранение данных (чтобы пережить смерть приложения)

Хранение конфигураций
Данные, которые завязаны на предпочтения пользователя (динамические темы)

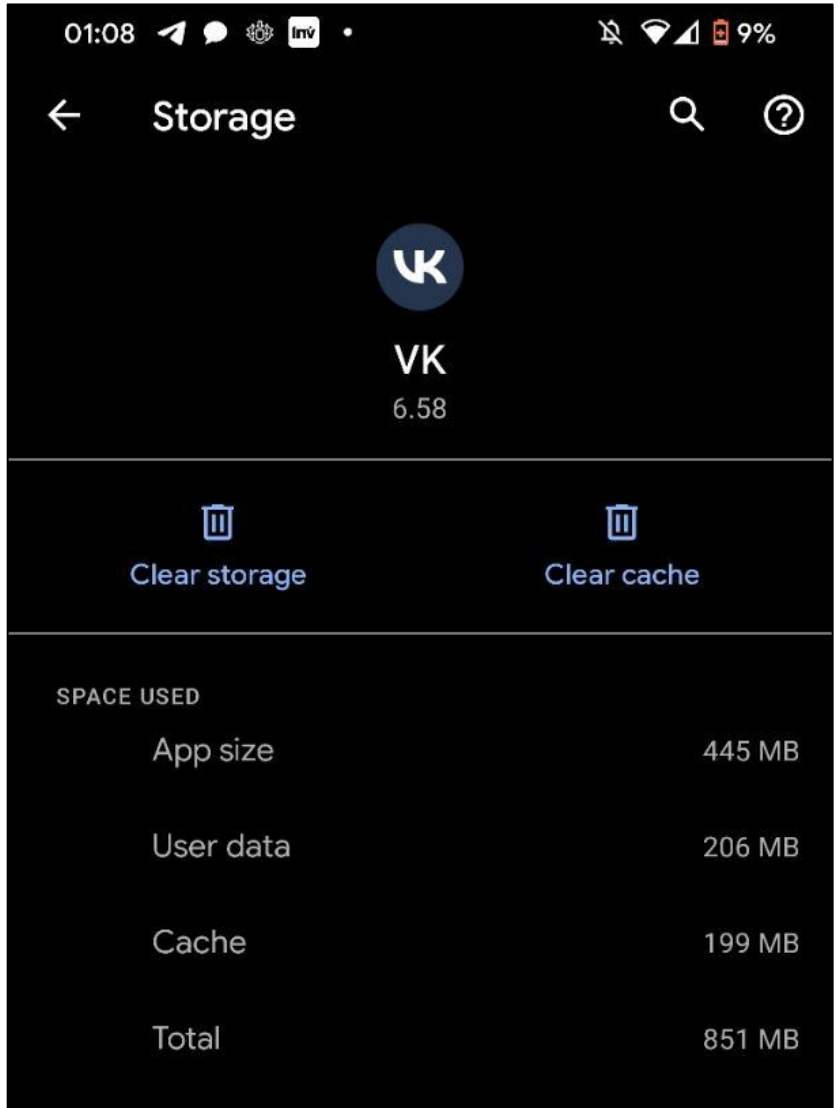
Пользовательский контент
UGC, отложенные загрузки

Хранение авторизационных данных
Токены, идентификаторы

Что-то еще…
???

Есть нюансы с хранением данных

- Пользователь может очистить кэш приложения
- Пользователь может стереть данные приложения
- Безопасность данных
- Данные бесконечно растут
- Обеспечения доступа к данным



savedInstanceState

Что знаете про этот тип хранения?

На случай, если было забыто

```
class MyFragment : Fragment() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)
    }

    override fun onSaveInstanceState(outState: Bundle) {
        super.onSaveInstanceState(outState)
    }
}
```

```
class MyActivity : Activity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
    }

    override fun onSaveInstanceState(outState: Bundle) {
        super.onSaveInstanceState(outState)
    }
}
```

Saved Instance State

Эти данные обычно используется для хранения состояния UI вашего экрана.

Но в этот “стэйт”, так же можно вкладывать и дополнительные параметры экрана (например, идентификаторы, или данные с которыми был открыт экран, или данные, которые экран приобрел для работы).

Несмотря на то, что в документации данные хранятся “в памяти”, при убивании вашего приложения в фоне, эти данные сохраняются в персистетнтом хранилище, в результате чего, хранящиеся внутри данные должны быть сериализуемы.

Доступ к нему вам предоставляет Activity и Fragment.

	ViewModel	Saved instance state
Storage location	in memory	in memory
Survives configuration change	Yes	Yes
Survives system-initiated process death	No	Yes
Survives user complete activity dismissal/onFinish()	No	No
Data limitations	complex objects are fine, but space is limited by available memory	only for primitive types and simple, small objects such as String
Read/write time	quick (memory access only)	slow (requires serialization/deserialization)

SharedPreferences

Shared Preferences

Интерфейс для доступа к хранению “примитивных данных”. Исходя из названия, часто использовалось именно для настроек.

Сохраняет данные в виде xml в приватном хранилище, из-за чего в чистом виде хранение “важных” данных в нем, считается небезопасным.

Управление данными происходит за счет Editor(для записи) Есть несколько видов “сохранения”:

- commit - синхронная записать
- apply - асинхронная запись

Для получения данных не требует Editor.

Может хранить только “примитивные данные” - String, Int, Float, Long, Boolean, Set<String>.

Хранилище может иметь несколько видов доступа, но обычно используется приватный вариант - Context.MODE_PRIVATE

```
// Инициализация
val preferences = context.getSharedPreferences(name, Context.MODE_PRIVATE)


// Чтение
preferences.getString(KEY_VALUE_PATTERN.format(index), "NONE")
preferences.getInt(KEY_COLOR_PATTERN.format(index), Color.BLACK)


// Запись, с "сахаром"
preferences.edit {
    putInt(KEY_COUNT, index)
    putString(KEY_VALUE_PATTERN.format(index), plate.value)
    putInt(KEY_COLOR_PATTERN.format(index), plate.color)
}
```

Как его сделать безопасным?

Добавление шифрования при записи/чтении

Сама идея заключается в следующем - в выборе алогритма шифровании. В результате - данные храним в шифрованом виде, а данные, необходимы для шифровании и расшифровке данных - храним отдельно.

Логично, что при шифровании и расшифровании будет увеличиваться время выполнения операций.

Если нет желания писать самим такой механизм - можно воспользоваться компонентом **EncryptedSharedPreferences** из androidx security-crypto библиотеки.

```
val masterKey = new MasterKey.Builder(context)
     .setKeyScheme(MasterKey.KeyScheme.AES256_GCM)
     .build()

 val sharedPreferences = EncryptedSharedPreferences.create(
     context,
     "secret_shared_prefs",
     masterKey,
     EncryptedSharedPreferences.PrefKeyEncryptionScheme.AES256_SIV,
     EncryptedSharedPreferences.PrefValueEncryptionScheme.AES256_GCM
 )

 // use the shared preferences and editor as you normally would
 val editor = sharedPreferences.edit();
```

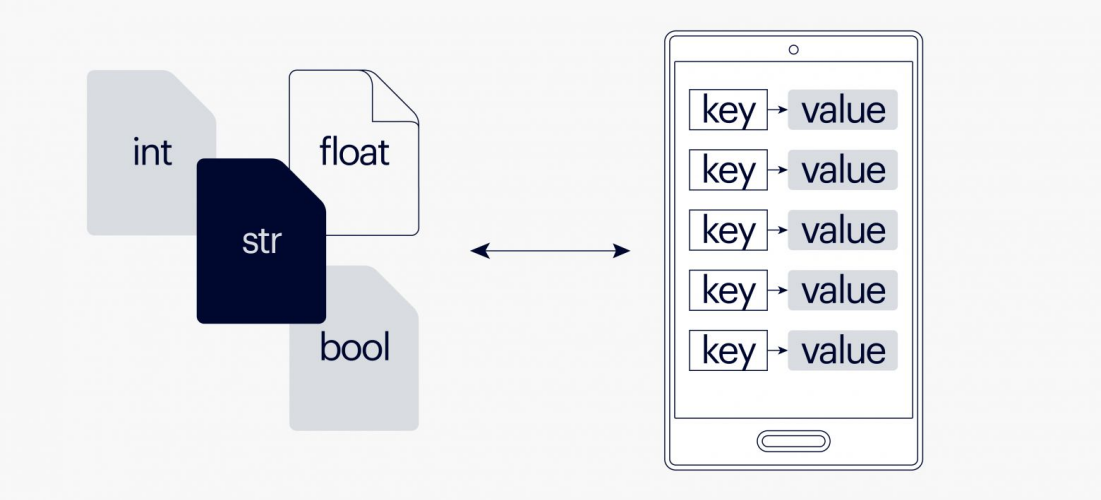
Альтернатива - DataStore

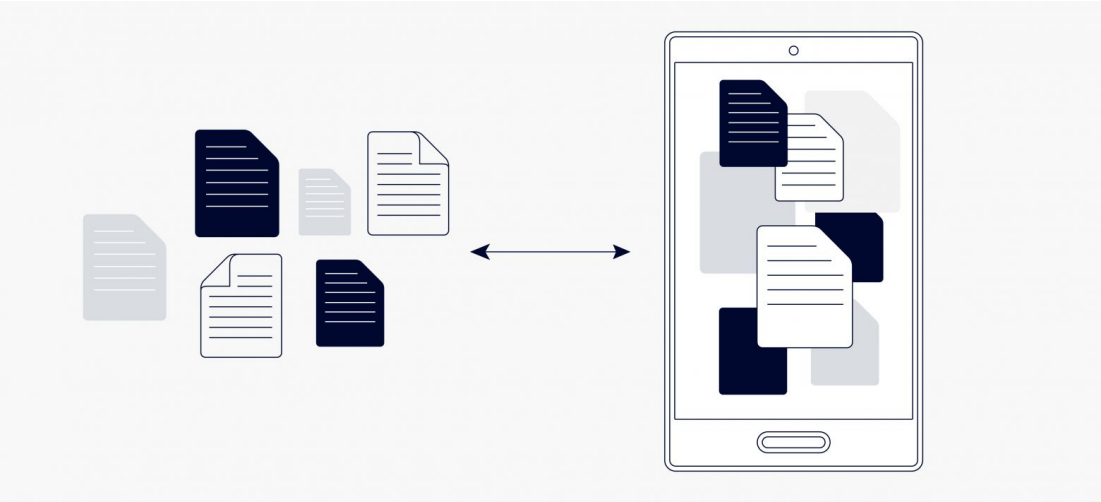
Официальная документация Google говорит о том, что SharedPreferences будет заменяться на DataStore. Но пока процесс замены происходит медленно.

DataStore сейчас - это отдельная библиотека из androidx. И можно сказать что она только недавно получила стабильный релиз.

Там есть два варианта имплементации:

- Preferences DataStore, на основе ключе-значение, как SharedPreferences
- Proto DataStore, хранит объекты. Но объекты надо будет описывать на основе протокола ProtoBuf, в отдельном файле.





Feature	SharedPreferences	PreferencesDataStore	ProtoDataStore
Async API	✅ (only for reading changed values, via listener)	✅ (via `Flow` and RxJava 2 & 3 `Flowable`)	✅ (via `Flow` and RxJava 2 & 3 `Flowable`)
Synchronous API	✅ (but not safe to call on UI thread)	❌	❌
Safe to call on UI thread	❌(1)	✅ (work is moved to `Dispatchers.IO` under the hood)	✅ (work is moved to `Dispatchers.IO` under the hood)
Can signal errors	❌	✅	✅
Safe from runtime exceptions	❌(2)	✅	✅
Has a transactional API with strong consistency guarantees	❌	✅	✅
Handles data migration	❌	✅	✅
Type safety	❌	❌	✅ with Protocol Buffers

Еще безопаснее?

Android Keystore System

Для безопасного хранения ключей на устройстве. Гарантирует сохранность материала, даже если процессы приложения будут скомпрометированы.

Можно управлять политикой использования ключей.

… но это лучше уже изучать в рамках безопасности приложения.

Для упрощения работы с таким хранилищем - лучше обратить внимание на фреймворк **google tink**.

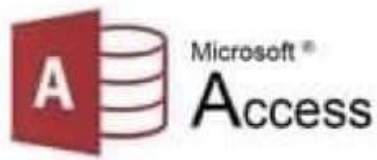
```
val kpg: KeyPairGenerator = KeyPairGenerator.getInstance(
        KeyProperties.KEY_ALGORITHM_EC,
        "AndroidKeyStore"
)
val parameterSpec: KeyGenParameterSpec = KeyGenParameterSpec
    .Builder(alias,KeyProperties.PURPOSE_SIGN or
KeyProperties.PURPOSE_VERIFY).run {
        setDigests(KeyProperties.DIGEST_SHA256, KeyProperties.DIGEST_SHA512)
        build()
    }

kpg.initialize(parameterSpec)
val kp = kpg.generateKeyPair()

/*
 * Verify a signature previously made by a private key in the
 * KeyStore. This uses the X.509 certificate attached to the
 * private key in the KeyStore to validate a previously
 * generated signature.
 */
val ks = KeyStore.getInstance("AndroidKeyStore").apply {
    load(null)
}
val entry = ks.getEntry(alias, null) as? KeyStore.PrivateKeyEntry
if (entry == null) {
    Log.w(TAG, "Not an instance of a PrivateKeyEntry")
    return false
}
val valid: Boolean = Signature.getInstance("SHA256withECDSA").run {
    initVerify(entry.certificate)
    update(data)
    verify(signature)
}
```

SQL


ORACLE
SQL Server
PostgreSQL
MySQL
SYBASE
IBM DB2.


Microsoft
Access


Excel


I PREFER A REAL DATABASE
I SAID A REAL DATABASE
PERFECTION

Hello SQLite

В Android из коробки есть одна база данных - это SQLIte.

Она имеет ряд ограничений, и ее версия зависит в основном от версии Android. В результате чего, официальная документация по возможностям DB может применяться не ко всем Android.

Но если не требуется сложных операций, то может быть достаточным использование базовых функций.

Традиционно считается, что DB быстрее отрабатывает при записи и поиске данных (время выполнения операции слабо зависит от количества данных за счет индексирования).

Для работы с SQLite традиционно используется SQLiteOpenHelper, а данные представляются в виде Cursor.

От разработчика будет требоваться знание SQL, и внутренних объектов Android, которые помогают работать с SQLite.

```
class PlateSqlHelper(context: Context) : SQLiteOpenHelper(context,
"SqlAccessor", null, 1) {
    override fun onCreate(db: SQLiteDatabase?) {
        if (db == null) {
            return
        }

        db.execSQL(CREATE_PLATE_TABLE)
    }

    override fun onUpgrade(db: SQLiteDatabase?, oldVersion: Int, newVersion:
Int) {
        if (db == null) {
            return
        }

        db.execSQL(DROP_PLATE_TABLE)
        onCreate(db)
    }

    fun plates(): List<Plate> {
        return readableDatabase.use { db ->
            db.rawQuery(SELECT_PLATE_ALL, null).use { cursor ->
cursor.convert() }
        }
    }
}
```

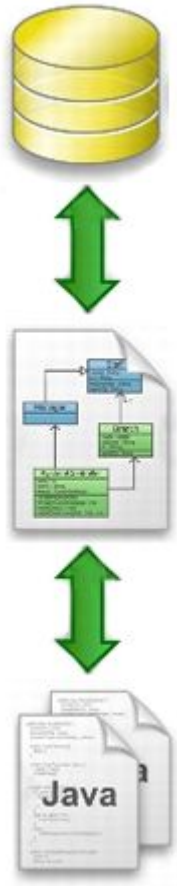
А можно как-то проще?

Hello ORM (object-relational mapping)

Объектно-реляционное сопоставление:)

Простыми словами - за счет высокоуровневого описания объектов, таблиц и связей, фреймворк за вас будет сам понимать, какие требуется делать запросы для получения/сохранения данных.

Примеры фреймворков: ORMLite, GreenDAO, OjectBox




```
// Пример использования ObjectBox
@Entity
data class User(
        @Id
        var id: Long = 0,
        var name: String? = null
)
...

try {
    box.put(User("Sam Flynn"))
} catch (e: UniqueViolationException) {
    // A User with that name already exists.
}
...

val query = box.query().run {
    equal(property, value)
    order(property)
    build()
}
```

Hello Room

Фреймворк от Google, который пишет запросы для SQLite за вас.

В 100% случаях за вас скрипт фреймворк не напишет, поэтому знания SQL желательны. Но для простых структур - достаточно знание работы самого фреймворка.

Работа фреймворка основывается на большом количестве аннотацию, при помощи которых описываются сущности и интерфейсы.

```
// Пример описания структуры без SQL
@Entity(
    tableName = "statistic",
    indices = [Index("id")],
    foreignKeys = [
        ForeignKey(
            entity = ResultsDbEntity::class,
            parentColumns = ["id"],
            childColumns = ["result_id"]
        ),
        ForeignKey(
            entity = DifficultyLevelsDbEntity::class,
            parentColumns = ["id"],
            childColumns = ["difficult_id"]
        )
    ]
)
data class StatisticDbEntity(
    @PrimaryKey(autoGenerate = true) val id: Long,
    @ColumnInfo(name = "result_id") val resultId: Long,
    @ColumnInfo(name = "difficult_id") val difficultId: Long,
    val mistakes: Long,
    val points: Long
)

@Entity(tableName = "results")
data class ResultsDbEntity(
    @PrimaryKey val id: Long,
    @ColumnInfo(name = "result_name") val resultName: String
)

@Dao
interface StatisticDao {
    @Insert(entity = StatisticDbEntity::class)
    fun insertNewStatisticData(statistic: StatisticDbEntity)

    @Query("SELECT statistic.id, result_name, difficulty_name, mistakes, points
FROM statistic\n" +
            "INNER JOIN results ON statistic.result_id = results.id\n" +
            "INNER JOIN difficulty_levels ON statistic.difficult_id =
difficulty_levels.id;")
    fun getAllStatisticData(): List<StatisticInfoTuple>

}
```

Hello NoSql

Не реляционная структура:)

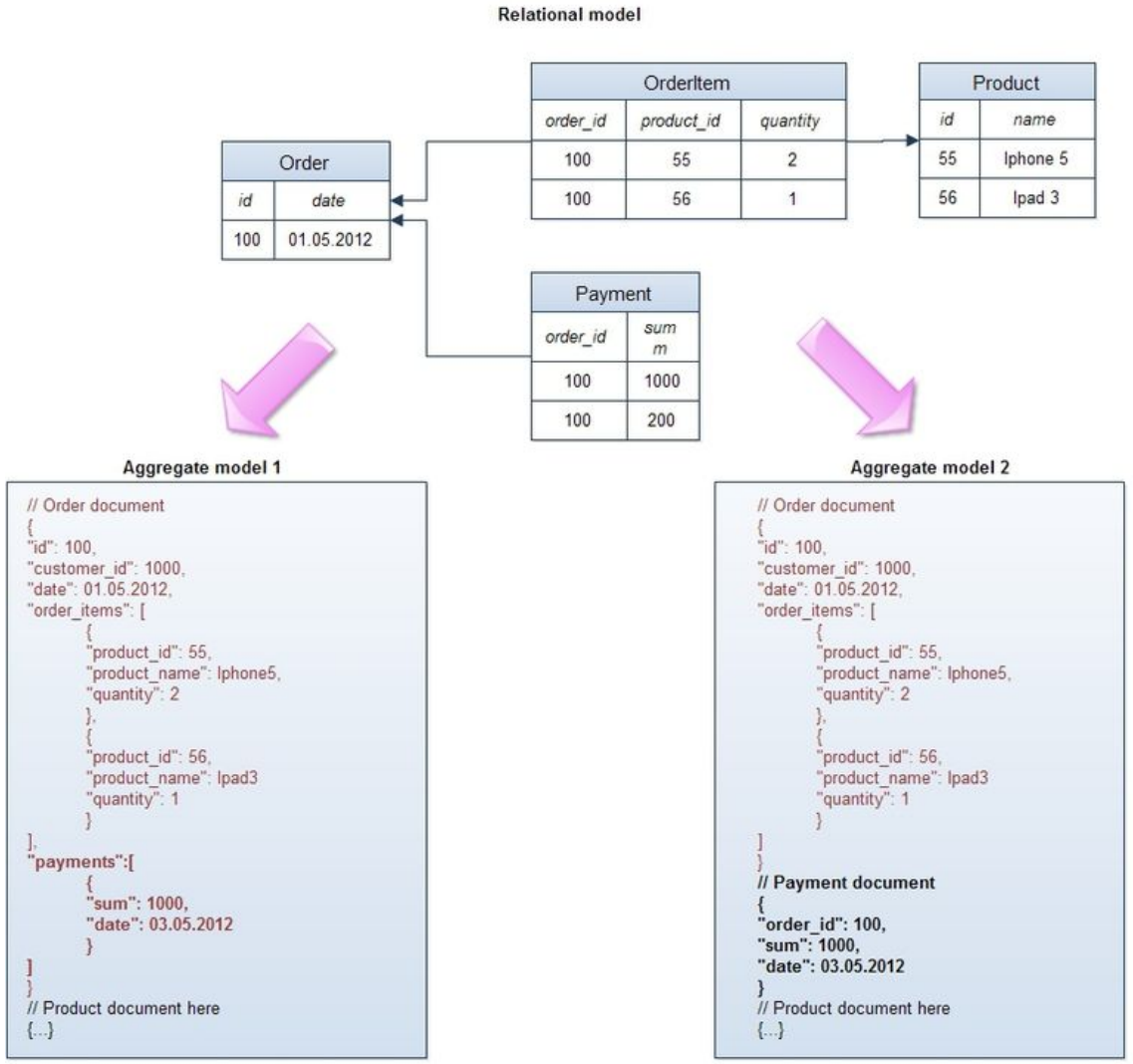
Для не супер сложных зависимостей - одно из лучших решений. Гибкая структура, минимальные описания структур. Для Android такие фреймворки вообще выглядят как key-value хранилища.

Вне Android тоже достаточно популярный подход. Как пример - MongoDB.

Примеры фреймворков: realm, OrmLite, PaperDB, Hawk

```
// Пример записи на PaperDB
Paper.book("for-user-1").write("contacts", contacts)
Paper.book("for-user-2").write("contacts", contacts)

val contacts = Paper.book().read("contacts")
```



Минусы и плюсы?

- Поскольку высокоуровневый язык использования - требуется изучать особенности фреймворка, чтобы разруливать сложные кейсы;
- Основная проблема зачастую - это перенос данных при обновлении схем баз данных;
- Исправление ошибок может потребовать много времени.

- Для простых ситуаций проблем не вызывает
- Не требует изучать ограничения SQLite;
- Если база в фреймворке вообще самописная, то может предоставлять больше возможностей, чем обычный SQLite;
- Все равно над SQLite будут писаться какие-то упрощения, а в ORM и прочих фреймворках эти упрощения уже и так написаны.

Немного историй

File Storage

External Storage

Для начала…

Необходимо проверить состояние внешнего хранилища:

- Environment.getExternalStorageState()

Возможные состояния:

- Environment.MEDIA_MOUNTED
- Environment.MEDIA_MOUNTED_READ_ONLY

Вншнее хранилище

Начиная с Android API 30 надо уже проверять разрешение, для получения свободного доступа:

- Environment.isExternalStorageManager()

Или использовать общие хранилища:

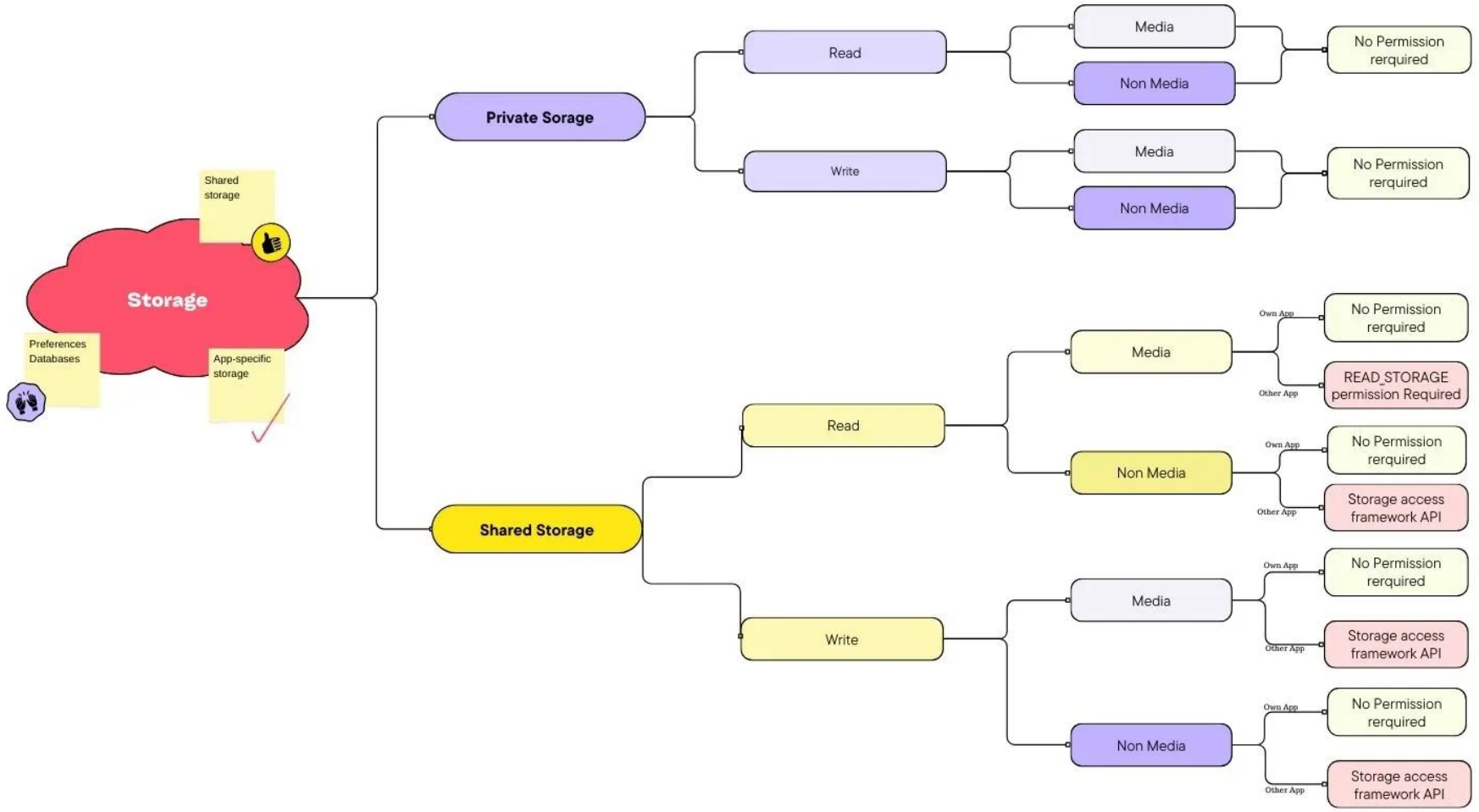
```
<uses-permission
    android:name="android.permission.WRITE_EXTERNAL_STORAGE"
    android:maxSdkVersion="29"
    />

<uses-permission
    android:name="android.permission.MANAGE_EXTERNAL_STORAGE"
    android:minSdkVersion="30"
    />
```

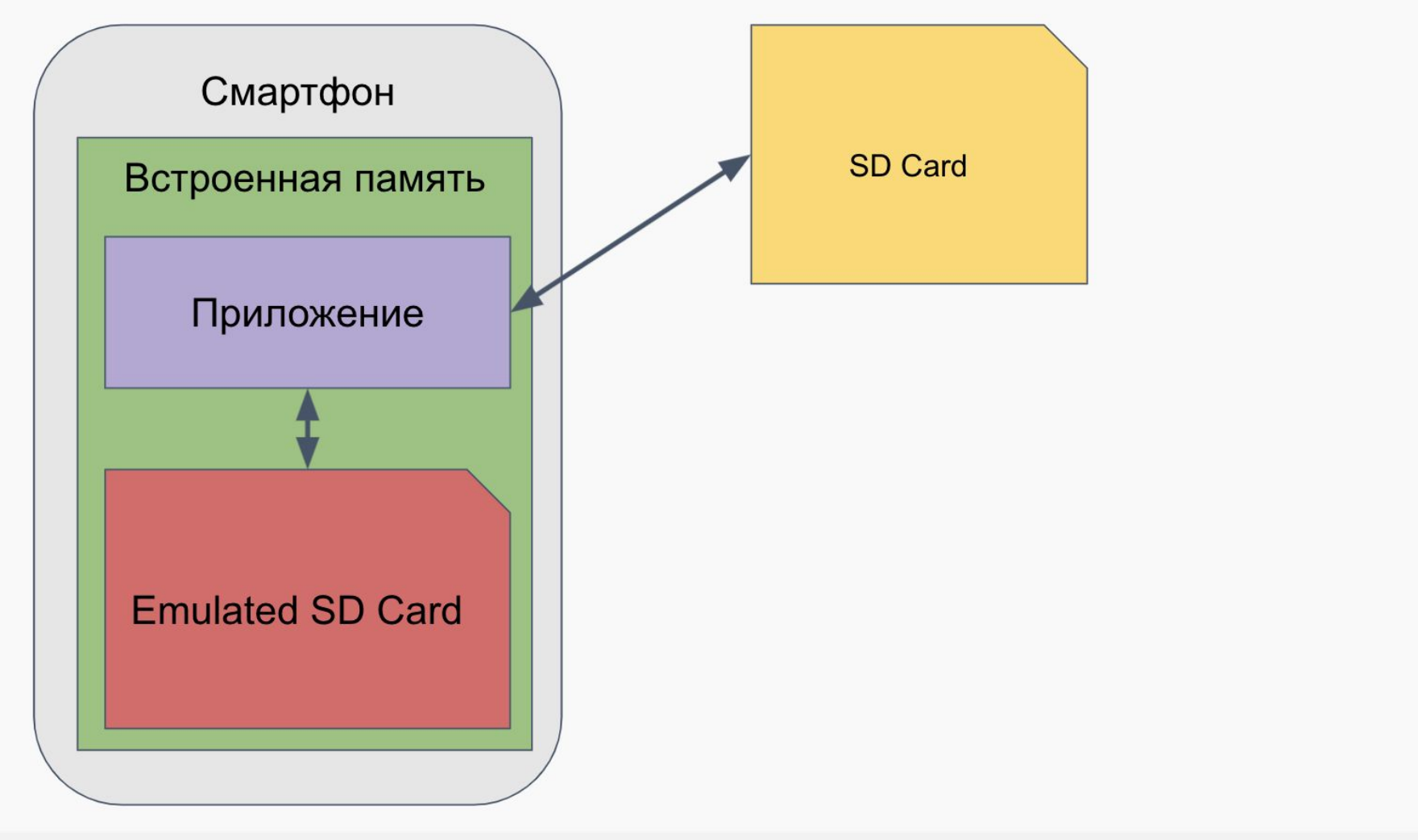
```
fun isExternalStorageWritable(): Boolean  {
    val state = Environment.getExternalStorageState()
    return Environment.MEDIA_MOUNTED == state
}

fun isExternalStorageReadable(): Boolean  {
    val state = Environment.getExternalStorageState()
    return Environment.MEDIA_MOUNTED == state
      || Environment.MEDIA_MOUNTED_READ_ONLY == state
}
```

Далее уже смотреть на Storage Access Framework



Внешнее хранилище, иногда не внешнее..

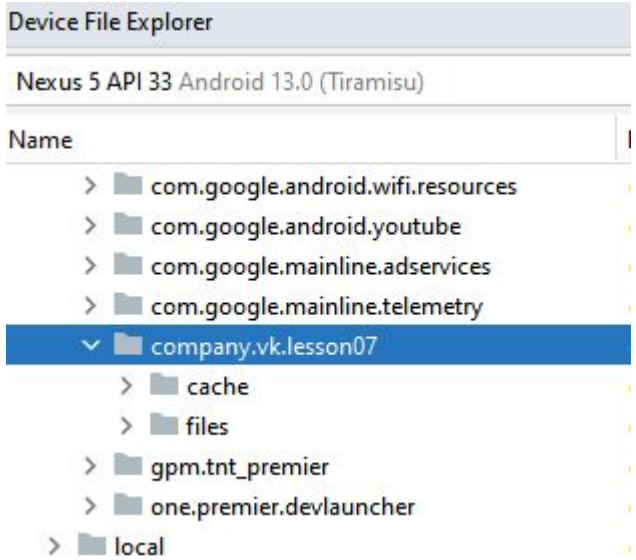


Internal Storage

Простыми словами - доступ к файлам имеет только ваше приложение. В общем случае - никто в эти файлы влезть не сможет.

Важные методы Context:

- getFilesDir() - путь до приватной директории приложения
- getDir(String name, int mode) - открывает/создает директорию в приватном хранилище
- getCacheDir() - путь до директории для хранения кэшей
- deleteFile(String name) - удаляет приватный файл
- fileList() - список приватных файлов



Account Manager

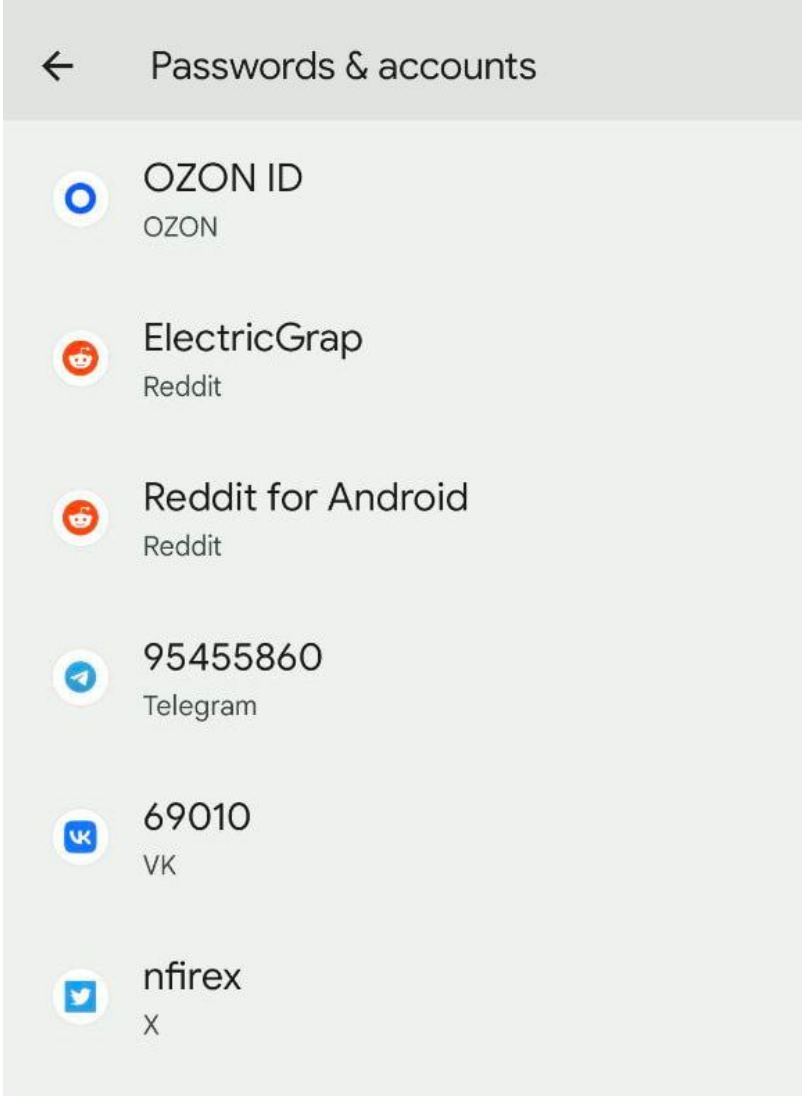
Хранение данных вне приложения

- Данные не теряются (есть НО)
- Тесно интегрирован в систему
- Сложная реализация!

Но, при правильной реализации, можно создать аккаунт, при помощи которого могут авторизоваться другие приложения (как пример - гугловые приложения).

Для работы этого подхода:

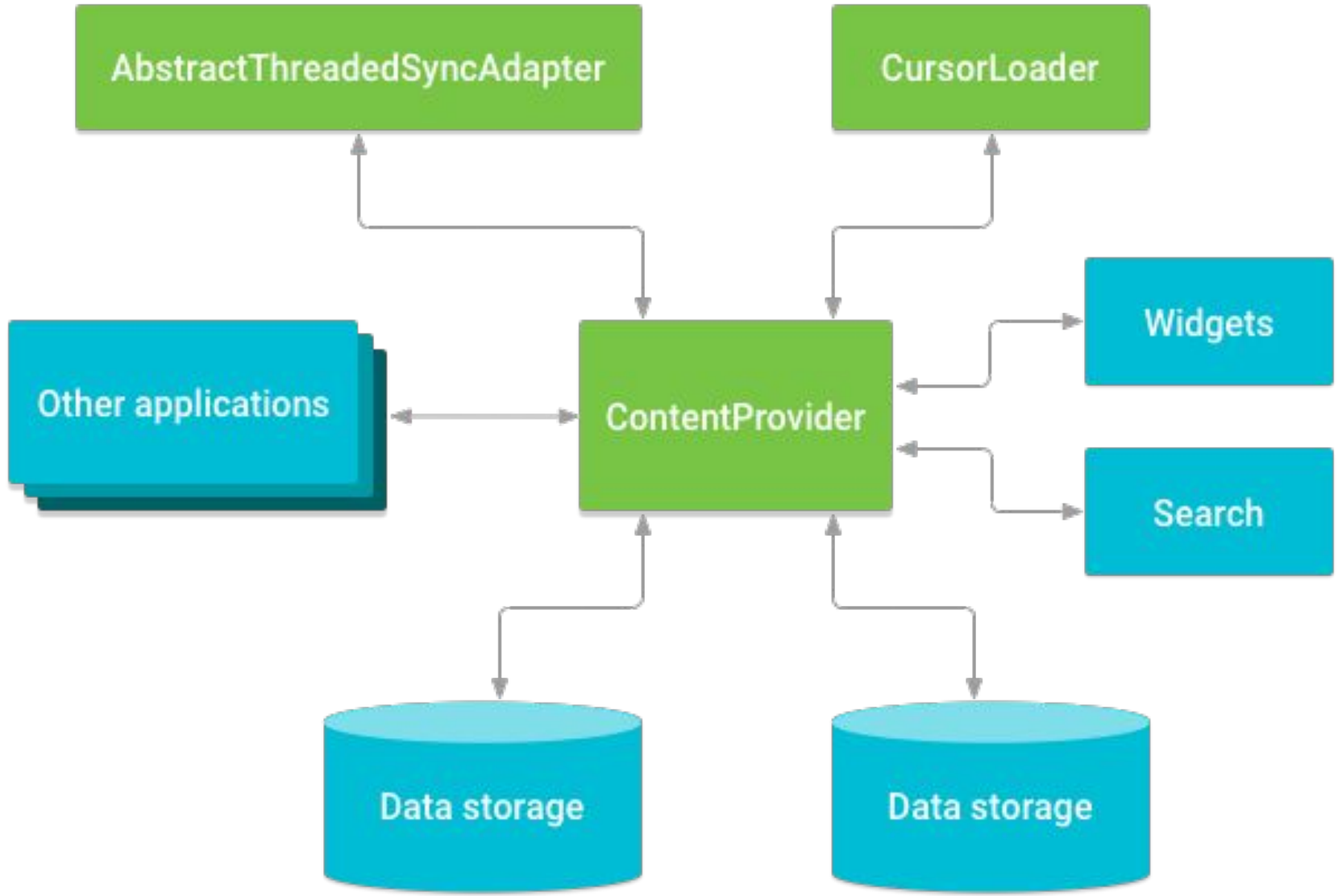
1. Зарегистрировать пермишены для работы с аккаунтами
2. Реализовать AbstractAccountAuthenticator
3. Реализовать Service для предоставления аутентификатора
4. Если необходима фоновое обновление данных - то еще реализовать и SyncAdapter
5. И фиксить возникающие проблемы:)



Content Provider

Основной компонент Android!

Старт приложения


AbstractThreadedSyncAdapter
CursorLoader
Other applications
ContentProvider
Widgets
Search
Data storage
Data storage

ContentProvider

Интерфейс для доступа к данным приложения. Как из своего приложения так и из другого.

Хранение данных остается на совести разработчика.

Инициализируется для одного процесса (?), но может данные передавать разным процессам, в виде Cursor.

Создается до Application.onCreate! И эту особенность можно использовать в разных целях.

Важно заполнить поля:

- **name** - сам класс с именем пакета
- **authorities** - идентификатор для поиска провайдера. Должен быть уникальным.

```
<provider android:authorities="list"
          android:directBootAware=["true" | "false"]
          android:enabled=["true" | "false"]
          android:exported=["true" | "false"]
          android:grantUriPermissions=["true" | "false"]
          android:icon="drawable resource"
          android:initOrder="integer"
          android:label="string resource"
          android:multiprocess=["true" | "false"]
          android:name="string"
          android:permission="string"
          android:process="string"
          android:readPermission="string"
          android:syncable=["true" | "false"]
          android:writePermission="string" >
    ...
</provider>
```

Имплементация... ох....

Необходимо имплементировать:

- **onCreate**() - вызывается при инициализации
- **query**(Uri, String, Bundle, CancellationSignal) - для возврата данных
- **insert**(Uri, ContentValues) - для записи данных
- **update**(Uri, ContentValues, Bundle) - для обновления данных
- **delete**(Uri, Bundle) - для удаления данных
- **getType**(Uri) - для определения MIME-типа данных

Взаимодействие идет через Context.contentResolver.

Основные методы:)

- **query** - для получения данных
- **insert** - для вставки данных
- **update** - для обновления данных
- **delete** - для удаления данных

Методы имеют вариативные параметры, которые управляют данными, сортировкам.

Еще раз вспомним про особенности

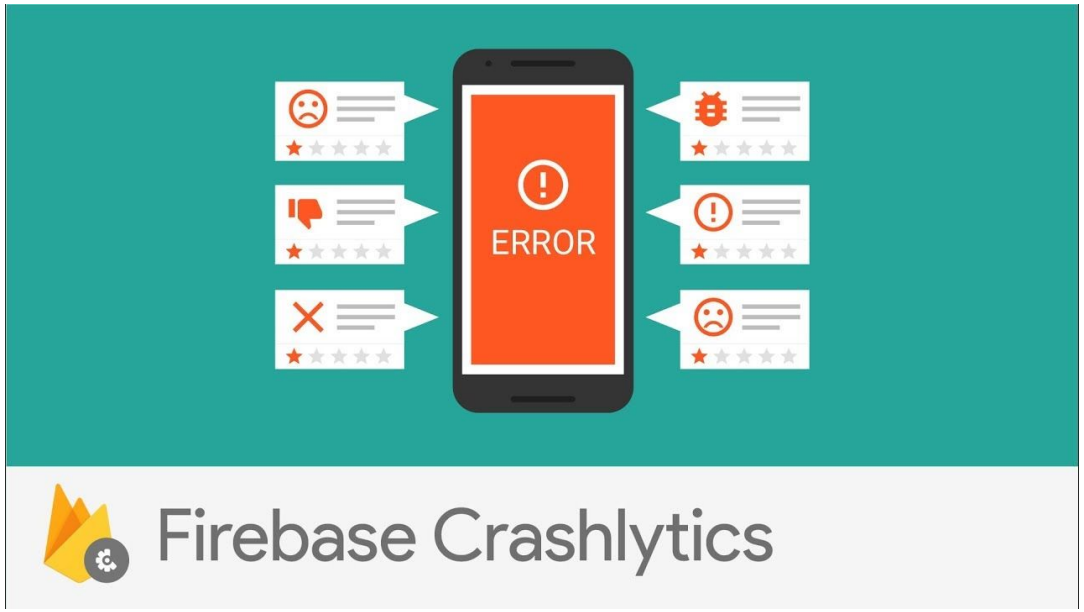
Запускается до старта приложения!

Эта особенность используется фремйворками для ранней инициализации:

- Фреймоврк для ловли крэшей
- Инициализация некоторых систем аналитик
- Инициализация компонентов

Обратить внимание на фреймворки:

- firebase crashlytics
- androidx App Startup



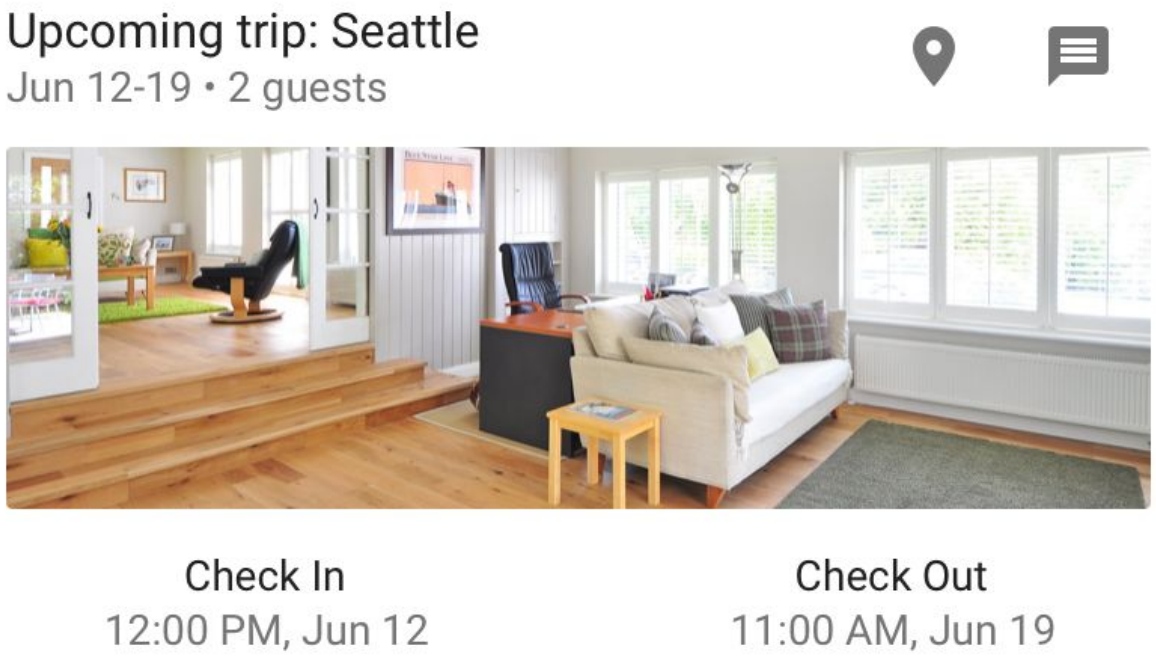
Еще интересное использование - SliceProvider

Предоставление системе контента в виде Слайсов.

Данный компонент скорее стоит воспринимать как пример развития ContentProvider. К сожалению, после анонса, Слайсы не были сильно востребованы, и сейчас в системе увидеть их не так просто.

```
public MySliceProvider extends SliceProvider {

     public Slice onBindSlice(Uri sliceUri) {
         String path = sliceUri.getPath();
         switch (path) {
             case "/weather":
                 return createWeatherSlice(sliceUri);
             case "/traffic":
                 return createTrafficSlice(sliceUri);
         }
         return null;
     }
}
```



Спасибо за внимание!