



План занятия

1. [Задача](#)
2. [Хранение данных](#)
3. [Shared Preferences](#)
4. [Файлы](#)
5. [Итоги](#)



ЗАДАЧА



ЗАДАЧА

Пришло время поговорить о хранении данных в Android. До этого мы обсудили с вами ряд вещей:

1. LifeCycle Activity и LifeCycle ViewModel
2. InstanceState (включая то, что состояние полей ввода Android сохраняет сам)

Сейчас наша задача: рассмотреть варианты персистентного (сохраняющегося между запусками приложения) сохранения данных и их использования.



ХРАНЕНИЕ ДАННЫХ

ХРАНЕНИЕ ДАННЫХ

По большому счёту, у нас не так много опций:

- SharedPreferences и аналоги
 - Файлы
 - Локальные базы данных
 - Удалённые сервисы (backend)
- рассмотрим сегодня

Часть из рассмотренных опций является просто надстройкой над файлами, но поскольку API значительно отличается, мы будем их рассматривать как отдельный вид.



SHARED PREFERENCES



SHARED PREFERENCES

[Shared Preferences](#) — API, позволяющее хранить key-value пары.

Позволяет хранить «примитивные» значения (под примитивными в данном случае понимаются числа, Boolean и строки).

ИСПОЛЬЗОВАНИЕ

В Activity (которое является наследником Context'a):

```
run { this: MainActivity
    val preferences = getPreferences(Context.MODE_PRIVATE)
    preferences.edit().apply { this: SharedPreferences.Editor!
        putString("key", "value") // putX
        commit() // commit - синхронно, apply - асинхронно
    } ^run
}
```

для записи нужен [Editor](#)

```
run { this: MainActivity
    getPreferences(Context.MODE_PRIVATE)
        .getString(key: "key", defValue: "no value")?.let { it: String
            Snackbar.make(binding.root, it, BaseTransientBottomBar.LENGTH_INDEFINITE)
                .show()
        }
}
```




RUN

Функция `run` позволяет нам запускать блоки кода, чтобы они не пересекались по видимости.

В данном случае мы её использовали только для демонстрации того, что в одном `Activity` мы сначала пишем что-то в `SharedPreferences`, затем читаем. Если убрать блок записи и перезапустить приложение, то мы увидим, что сохранённое значение прочиталось.

MODE_PRIVATE

```
/**
 * File creation mode: the default mode, where the created file can only
 * be accessed by the calling application (or all applications sharing the
 * same user ID).
 */
public static final int MODE_PRIVATE = 0x0000;
```

```
/** File creation mode: allow all other applications to have read access to ...*/
```

→ @Deprecated

```
public static final int MODE_WORLD_READABLE = 0x0001;
```

```
/** File creation mode: allow all other applications to have write access to ...*/
```

→ @Deprecated

```
public static final int MODE_WORLD_WRITEABLE = 0x0002;
```

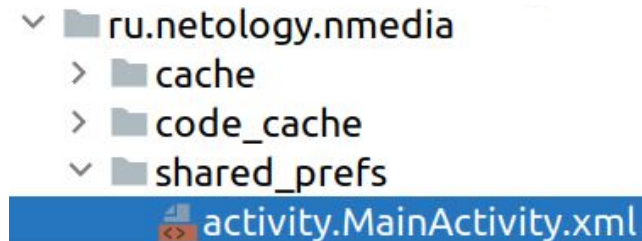
FILE

В дебаггере можно увидеть, где на самом деле хранятся Shared Preferences:

```
✓ ☰ preferences = {SharedPreferencesImpl@16605}
  f mBackupFile = {File@16653} "/data/user/0/ru.netology.nmedia/shared_prefs/activity.MainActivity.xml.bak"
  f mCurrentMemoryStateGeneration = 0
  f mDiskStateGeneration = 0
  f mDiskWritesInFlight = 0
  f mFile = {File@16654} "/data/user/0/ru.netology.nmedia/shared_prefs/activity.MainActivity.xml"
```

DEVICE FILE EXPLORER

В Android есть специальный инструмент Device File Explorer (Shift + Shift и ищите, либо меню View Tool Window), который позволяет вам просмотреть файловую систему устройства:



```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
  <string name="key">value</string>
</map>
```

Единственное, в дебаггере и в Device File Explorer'e пути будут немного отличаться (нужно смотреть /data/data/ru.netology.nmedia)



SHARED PREFERENCES

При использовании подобного подхода на каждое Activity будет создаваться свой файл.

Можно создать общий файл, вызывая `getSharedPreferences` вместо `getPreferences`.

SHARED PREFERENCES

Попробуем использовать SharedPreferences как постоянное хранилище для нашего репозитория (в реальной жизни вы так делать не будете, но нам это нужно для обсуждения концепции).

Итак, вопросов в этой идее несколько:

1. Посты — это объекты, а SharedPreferences позволяет хранить только «примитивы».
2. Для работы с SharedPreferences нужен Context, а у репозитория (и ViewModel, которая взаимодействует с ним) контекста нет.



СЕРИАЛИЗАЦИЯ

Одна из идей, которую мы будем использовать на протяжении всей профессии — сериализация. Мы можем преобразовать наши данные в какой-то другой формат, например, строки.

Конечно же, вручную это делать не очень интересно, поэтому мы воспользуемся библиотекой и готовым форматом.



JSON

[JSON](#) (JavaScript Object Notation) — один из самых популярных платформонезависимых форматов данных.

Используется как для хранения структурированных данных, так и для обмена данными (например, между Android приложением и backend'ом).

GSON

[GSON](#) — Java библиотека, позволяющая конвертировать объекты в JSON (и обратно).

```
implementation "com.google.code.gson:gson:$gson_version"
```

CONTEXT

Теперь самый важный вопрос: Context. Мы с вами знаем, что Activity является Context'ом, так почему бы не передать (обычно говорят «прокинуть») его во ViewModel, а из ViewModel в Repository?

Это очень плохая идея: никогда не передавайте Activity, View или Fragment (познакомимся на следующей лекции) во ViewModel — получите утечку памяти (т.к. Lifecycle ViewModel и Lifecycle описанных сущностей может отличаться).

ANDROIDVIEWMODEL

Когда вам в Architectural Components требуется Context, наследуйтесь от AndroidViewModel:

```
class PostViewModel(application: Application) : AndroidViewModel(application) {  
    // упрощённый вариант  
    private val repository: PostRepository = PostRepositorySharedPrefsImpl(application)  
    val data = repository.getAll()  
    val edited = MutableLiveData(empty)  
  
    fun save() {...}  
  
    fun edit(post: Post) {...}  
  
    fun changeContent(content: String) {...}  
  
    fun likeById(id: Long) = repository.likeById(id)  
    fun removeById(id: Long) = repository.removeById(id)  
}
```



APPLICATION

Application — это базовый класс для управления глобальным состоянием приложения (также является контекстом, но про него говорят, что это контекст приложения).

По факту, экземпляр именно этого класса создаётся при создании процесса вашего приложения.

Добраться до этого контекста из любого другого контекста можно с помощью вызова `getApplicationContext`.

```

class PostRepositorySharedPrefsImpl(
    context: Context,
) : PostRepository {
    private val gson = Gson()
    private val prefs = context.getSharedPreferences(name: "repo", Context.MODE_PRIVATE)
    private val type = TypeToken.getParameterized(List::class.java, Post::class.java).type
    private val key = "posts"
    private var nextId = 1L
    private var posts = emptyList<Post>()
    private val data = MutableLiveData(posts)

    init {
        prefs.getString(key, defValue: null)?.let { it: String
            posts = gson.fromJson(it, type)
            data.value = posts
        }
    }

    // для презентации убрали пустые строки
    override fun getAll(): LiveData<List<Post>> = data
    override fun save(post: Post) {...}
    override fun likeById(id: Long) {...}
    override fun removeById(id: Long) {...}
    private fun sync() {
        with(prefs.edit()) { this: SharedPreferences.Editor!
            putString(key, gson.toJson(posts))
            apply()
        }
    }
}

```

TYPE

Отдельно стоит отметить следующую строку:

```
private val type = TypeToken.getParameterized(List::class.java, Post::class.java).type
```

Она нужна только для того, чтобы объяснить GSON, что мы хотим получить List из Post (поскольку в самом JSON'е этой информации нет*).

Примечание*: увидим это дальше.

SYNC

И конечно же, в каждом методе изменения надо запускать синхронизацию:

```
override fun likeById(id: Long) {  
    posts = posts.map { it: Post  
        if (it.id != id) it else it.copy( ^map  
            likedByMe = !it.likedByMe,  
            likes = if (it.likedByMe) it.likes - 1 else it.likes + 1  
        ) ^map  
    }  
    data.value = posts  
    sync()  
}  
  
override fun removeById(id: Long) {  
    posts = posts.filter { it.id != id }  
    data.value = posts  
    sync()  
}
```

XML

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
  <string name="posts">
    [{"author":"Me","content":"demo","id":1,"likedByMe":false,"likes":0,"published":"now"}]
  </string>
</map>
```

Конечно же, возникает вопрос: почему бы сразу не хранить всё в файле, зачем нам лишняя надстройка в виде Shared Preferences?



КЛЮЧЕВОЕ

Благодаря архитектуре с разделением слоёв, мы внесли изменения только во ViewModel (отнаследовались от другого базового класса и указали другой репозиторий*) и в реализацию конкретного репозитория. Ни Activity, ни Adapter'ы, ни другие классы нам менять не пришлось.

Примечание*: только потому, что мы создаём Repository внутри ViewModel, чуть позже мы научимся подставлять туда реализацию и даже ViewModel меняться не будет.



JETPACK DATASTORE

Google представила новую библиотеку [Jetpack DataStore](#), которая призвана заменить Shared Preferences и позволяет хранить типизированные данные.

Но на данный момент библиотека еще не распространена, и мы пока рассматриваем самый популярный вариант.



ФАЙЛЫ



INTERNAL STORAGE vs EXTERNAL STORAGE

Для хранения файлов Android предоставляет два хранилища:

- Internal — защищённое от других приложений хранилище (шифруется начиная с Android 10).
- External — хранилище, в которое можно организовать доступ других приложений.

Оба хранилища очищаются при удалении приложений (так же, как и Shared Preferences).



DATA vs CACHE

Для каждого из хранилищ также есть разделение на Data (данные) и Cache (временные файлы, используемые для ускорения доступа).

Кэш нужен для того, чтобы, например, не перекачивать аватарки пользователей при просмотре ленты новостей.



DATA vs CACHE

Получить доступ к каждому из каталогов можно с помощью соответствующего property:

- filesDir
- cacheDir

О РАБОТЕ С ФАЙЛАМИ

Работа с файлами основана на нескольких ключевых концепциях:

- пути
- файлы
- потоки



ПУТИ

Путь — это строка, позволяющая путём навигации по файловой системе добраться до файла.

Причём сам путь тоже можно модифицировать с помощью соответствующих абстракций, например, можно взять путь до каталога и относительно этого пути получить путь до файла:

```
context.filesDir.resolve("posts.json")
```




ПУТИ

Пути бывают относительными и абсолютными.

Абсолютный путь — это путь от корня файловой системы - / в Android.












Т.е. `/data/user/0/` — это абсолютный путь.

Относительные пути — это пути относительно какого-то уже существующего пути. Т.е. `posts.json` — это относительный путь.

ФАЙЛЫ

Файл — это абстракция, позволяющая манипулировать файлами: проверять существование, открывать, удалять и т.д.

Для нас основные манипуляции с путями и файлами представлены Java-классом `File` и extension-функциями к нему:

- ▼  `Utils.kt`
 - f  `copyRecursively(File, Boolean = ..., (File, IOException) -> OnErrorAction = ...) on File: Boolean`
 - f  `copyTo(File, Boolean = ..., Int = ...) on File: File`
 - f  `createTempDir(String = ..., String? = ..., File? = ...): File`
 - f  `createTempFile(String = ..., String? = ..., File? = ...): File`
 - f  `deleteRecursively() on File: Boolean`
 - f  `endsWith(File) on File: Boolean`
 - f  `endsWith(String) on File: Boolean`
 - ▼  `extension: String on File`
 - ▼  `invariantSeparatorsPath: String on File`
 - ▼  `nameWithoutExtension: String on File`

...

ПОТОКИ (STREAMS)

Любой набор байт, будь то файл, или байты, приходящие по сетевому соединению, можно представить в виде специальной абстракции — поток (Stream).

Существуют потоки, из которых можно читать (InputStream) и в которые можно писать (OutputStream).

Потоки рассматривают любые данные в виде последовательности байт и для них совершенно не важно, что это за данные (например, в какой кодировке, если мы используем строки).



READER & WRITER

Но если мы хотим иметь дело не с байтами, а с текстом (а именно его «хочет» GSON и его же выдаёт на выходе), то нам недостаточно просто байт.

Нам нужно к этим байтам применить кодировку (таблицу, сопоставляющую байты буквам) и читать уже по символам.

Для этого используются Reader'ы (чтение) и Writer'ы (запись) соответственно.



БУФЕРИЗАЦИЯ

Когда производится чтение или запись файлов, запись/чтение по одному байту является неэффективной. Поэтому используют буферизованные потоки/reader'ы/writer'ы, которые вычитывают/накапливают перед записью байт больше, чем нужно.



РЕСУРСЫ

И последнее, стоит закрывать файлы (вызывать метод `close`, описанный в интерфейсе `Closable`), несмотря на то, что Android закроет их при завершении процесса.



КОД

Конечно же, в Android и Kotlin есть уже куча готовых хэлперов, которые позволяют оформить это в типовые блоки.

```
class PostRepositoryFileImpl(
    private val context: Context,
) : PostRepository {
    private val gson = Gson()
    private val type = TypeToken.getParameterized(List::class.java, Post::class.java).type
    private val filename = "posts.json"
    private var nextId = 1L
    private var posts = emptyList<Post>()
    private val data = MutableLiveData(posts)

    init {
        val file = context.filesDir.resolve(filename)
        if (file.exists()) {
            // если файл есть - читаем
            context.openFileInput(filename).bufferedReader().use { it: BufferedReader
                posts = gson.fromJson(it, type)
                data.value = posts
            }
        } else {
            // если нет, записываем пустой массив
            sync()
        }
    }
}
```


ПО ШАГАМ

1. `context.filesDir.resolve` получает File с нужным путём
2. `file.exists()` проверяет существует ли файл (иначе при попытке чтения получим исключение)
3. `context.openFileInput` открывает InputStream
4. `.bufferedReader` позволяет получить буферизованный Reader
5. `.use` позволяет автоматически закрыть ресурс после использования (будет закрыт Reader, который сам закроет InputStream)
6. GSON умеет читать данные из Reader'a

SYNC

```
private fun sync() {  
    context.openFileOutput(filename, Context.MODE_PRIVATE).bufferedWriter().use { it: BufferedWriter  
        it.write(gson.toJson(posts))  
    }  
}
```



ПО ШАГАМ

1. `context.openFileOutput` открывает `OutputStream`
2. `.bufferedWriter` позволяет получить буферизованный `Writer`
3. `.use` позволяет автоматически закрыть ресурс после использования (будет закрыт `Writer`, который сам закроет `OutputStream`)
4. `it.write` записывает данные

Важно: если файла не было, он будет создан



ВАЖНО

В данном примере мы не обрабатываем исключения. Вполне возможно, что по каким-то причинам файл окажется «битым» и GSON не сможет его распарсить.

Поэтому попытку чтения нужно завернуть в try-catch и выбрать стратегию для обработки исключения (например, перезаписать файл пустыми данными — тогда пользователь потеряет все свои посты).



КРИПТОГРАФИЯ

Для всех рассмотренных нами вариантов хранения данных возможно использование криптографии (шифрования данных) для обеспечения их безопасности (сохранения конфиденциальности и целостности).

Вопросы безопасности будут рассматриваться на следующих курсах нашей профессии.



ИТОГИ

ИТОГИ

Сегодня мы обсудили вопросы хранения постоянных данных. Несмотря на то, что оба варианта решают нашу задачу, они не являются масштабируемыми, а тем более удобными. Но знать о них нужно (особенно о работе с файлами, включая просмотр файлов на устройстве).

Кроме того, если у вас данные небольшого объёма, то вы вполне можете эти варианты использовать (при этом крайне желательно просмотреть лекции по многопоточности, чтобы использовать вариант с обычным JSON безопасно).



В РЕАЛЬНОЙ ЖИЗНИ

В реальной жизни большинство приложений подключено к backend'у с помощью сети Интернет и даже набранные, но не отправленные сообщения сохраняются на нём.

Поэтому наша следующая задача — рассматривать именно такую схему, потому что для нас важно получение навыков реализации самых востребованных схем.