



План занятия

1. [Задача](#)
2. [Процессы и потоки](#)
3. [Управление состоянием](#)
4. [Single Live Event](#)
5. [Итоги](#)
6. [Домашнее задание](#)



ЗАДАЧА



ЗАДАЧА

Мы изучили клиент-серверную модель и теперь можем применить данные знания на практике.

Возьмём наш Android клиент и попробуем отправить запрос на сервер.

ОКНТТР

Для отправки запроса на сервер воспользуемся самой популярной библиотекой [okhttp](#)*.

Подключим её в `app/build.gradle`:

```
def okhttp_version = "4.9.1"  
implementation "com.squareup.okhttp3:okhttp:$okhttp_version"
```

Примечание*: можно делать запросы с помощью встроенных клиентов, но okhttp гораздо удобнее в использовании. Начиная с 4 версии, библиотека была переписана на Kotlin.

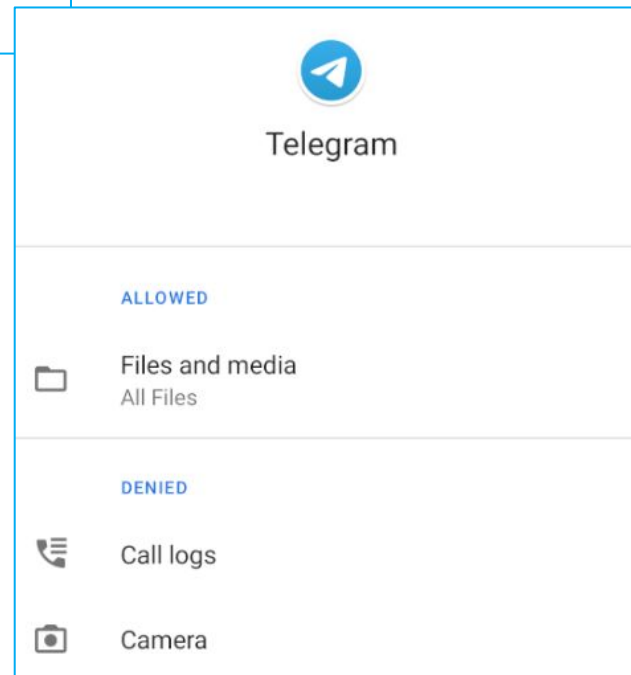
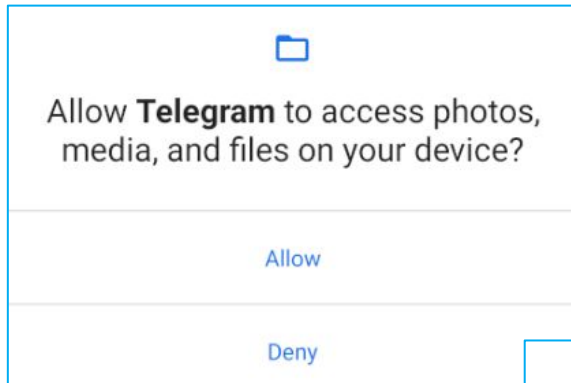
PERMISSIONS

Для запросов в сеть (за пределы устройства) нам нужны соответствующие разрешения ([permissions](#)) от пользователя.

Permissions (для разработчика) делятся на следующие группы:

1. Install-time (пользователю показываются при установке — требуют лишь записи в манифесте)
2. Runtime (запрашиваются у пользователя в момент использования — требуют кода для обработки решения пользователя)
3. Special (для производителей)

PERMISSIONS



PERMISSIONS

Кроме того, Permissions делятся по степени «опасности» (protection levels):

- normal
- dangerous
- signature
- privileged
- development

INTERNET

```
public static final String INTERNET
```

Allows applications to open network sockets.

Protection level: normal

Constant Value: "android.permission.INTERNET"

INTERNET

Доступ в сеть является Install-time permission, достаточно его прописать в [AndroidManifest.xml](#):

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="ru.netology.nmedia">

    <uses-permission android:name="android.permission.INTERNET" />

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="NMedia"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
```





HTTPS

Начиная с [Android 9, мы имеем право](#) использовать только защищённые (HTTPS) соединения.

Но HTTPS (как часто бывает при разработке) у нас пока нет, поэтому нам нужно явно указать в манифесте, что мы будем использовать HTTP.

BUILD TYPES

Проект в Android по умолчанию имеет два варианта сборки приложения:

- debug (для отладки)
- release (для публикации)

Build Variants  	
Module	Active Build Variant
 NMedia.app	debug

HTTP

Чтобы реализовать доступ к HTTP трафику только в debug сборке, мы воспользуемся механизмом [manifest placeholders](#):

```
buildTypes {
    release {
        minifyEnabled false
        proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'
        manifestPlaceholders.usesCleartextTraffic = false
    }
    debug {
        manifestPlaceholders.usesCleartextTraffic = true
    }
}
```

```
<application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="NMedia"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/AppTheme"
    android:usesCleartextTraffic="${usesCleartextTraffic}">
```



REPOSITORY

Для выполнения запроса нам понадобится OkHttpClient и Gson для десериализации (см.следующий слайд):

REPOSITORY

```
class PostRepositoryImpl: PostRepository {
    private val client = OkHttpClient.Builder()
        .connectTimeout(timeout: 30, TimeUnit.SECONDS)
        .build()
    private val gson = Gson()
    private val typeToken = object : TypeToken<List<Post>>() {}

    companion object {
        private const val BASE_URL = "http://10.0.2.2:9999"
        private val jsonType = "application/json".toMediaType()
    }

    override fun getAll(): List<Post> {
        val request: Request = Request.Builder()
            .url(url: "${BASE_URL}/api/slow/posts")
            .build()

        return client.newCall(request)
            .execute()
            .let { it.body?.string() ?: throw RuntimeException("body is null") }
            .let { it: String?
                gson.fromJson(it, typeToken.type)
            }
    }
}
```

СЕРВЕР NMEDIA

Напоминаем, что у нас есть сервер, который вам необходимо запускать на вашей локальной машине с помощью команд `./gradlew bootRun` Linux/Mac `.\gradlew bootRun` Windows в командной строке или через IDEA.

Сервер запускается на порту 9999 и доступен из Android эмулятора по адресу <http://10.0.2.2:9999>.

У сервера есть два пути:

- `/api` — обычные запросы (отрабатывают быстро)
- `/api/slow` — медленные запросы (с задержкой)

СЕРВЕР NMEDIA

Сервер поддерживает ключевые необходимые нам операции:

Получение списка сообщений

► **GET** <http://localhost:9999/api/posts>

Создание нового сообщения (id = 0)

► **POST** <http://localhost:9999/api/posts>
Content-Type: application/json

```
{
  "id": 0,
  "author": "Netology",
  "content": "First Post",
  "published": 0,
  "likedByMe": false,
  "likes": 0
}
```

СЕРВЕР NMEDIA

Получение сообщения по id

- ▶ GET <http://localhost:9999/api/posts/1>

Обновление сообщения (id != 0)

- ▶ POST <http://localhost:9999/api/posts>
Content-Type: application/json

```
{  
  "id": 1,  
  "author": "Netology",  
  "content": "First Post (UPDATED)",  
  "published": 0,  
  "likedByMe": false,  
  "likes": 0  
}
```

Удаление сообщения по id

- ▶ DELETE <http://localhost:9999/api/posts/1>

ПРОБЛЕМА

Пробуем запустить код:

Caused by: java.lang.reflect.InvocationTargetException

at java.lang.reflect.Constructor.newInstance0(Native Method) <1 internal call>

at androidx.lifecycle.ViewModelProvider\$AndroidViewModelFactory.create(ViewModelProvider.java:267)

Caused by: android.os.NetworkOnMainThreadException

E/AndroidRuntime: at android.os.StrictMode\$AndroidBlockGuardPolicy.onNetwork(StrictMode.java:1605)

Q: В чём может быть проблема?

The exception that is thrown when an application attempts to perform a networking operation on its main thread.

This is only thrown for applications targeting the Honeycomb SDK or higher. Applications targeting earlier SDK versions are allowed to do networking on their main event loop threads, but it's heavily discouraged. See the document [Designing for Responsiveness](#).

Also see [StrictMode](#).

Давайте разбираться, что такое потоки и о чём здесь идёт речь.



ПРОЦЕССЫ И ПОТОКИ

ПРОЦЕСС

Каждое Android-приложение существует в виде процесса ОС (так же как на вашей машине с Windows есть процессы, так и в Android есть процессы).

Процесс — это контейнер, определяющий:

- доступную память
- используемые ресурсы (открытые файлы, сетевые соединения)
- настройки безопасности
- потоки исполнения кода



ПОТОК

Поток (thread) – это путь программного выполнения. В каждом приложении есть как минимум 1 поток (главный - main).


Поток — это то, что может исполняться на процессоре устройства, а именно на его ядрах.



ПОТОК

Потоков может быть несколько сотен в рамках одного приложения, поэтому ОС занимается тем, что выделяет каждому потоку квант времени для выполнения, после чего приостанавливает поток и даёт выполниться другому.

Переключения происходят настолько быстро, что создаётся эффект параллельности (на самом деле параллельность может быть обеспечена только на многоядерных процессорах).



MAIN (UI) THREAD

Поток, который запускается при старте приложения, называют главным или **UI (User Interface) потоком**.

С версии Lolipop за UI отвечают 2 потока. Второй называется **RenderThread** и отвечает за плавные анимации. Однако, с точки зрения разработчика, такого разделения не существует, и мы будем употреблять термин UI поток в единственном числе.



МНОГОПОТОЧНОСТЬ

Проблема заключается в том, что мы не можем просто так выполнять продолжительные операции, т.к. это может привести к «зависанию» пользовательского интерфейса.

Сетевые операции являются отличным примером. С определённого момента развития Android SDK при выполнении запроса в сеть приложение аварийно завершает свою работу. Аналогичное поведение мы уже встречали при работе с библиотекой Room.

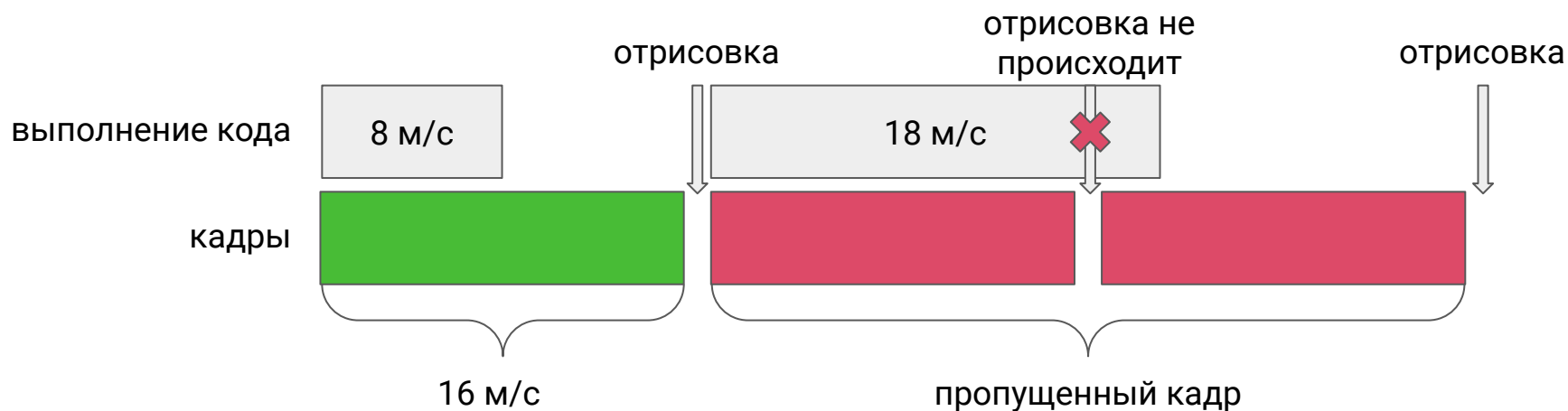
Из этого мы можем сделать вывод: запросы в сеть и в БД (файлы), нужно выполнять в другом потоке.



МНОГОПОТОЧНОСТЬ

Если пренебрегать данными советами, то могут возникать пропущенные кадры.

В большинстве устройств на данный момент частота обновлений экрана является 60 кадров в секунду. Таким образом, каждый кадр должен быть выполнен за 16 и менее миллисекунд.



ЖИЗНЕННЫЙ ЦИКЛ UI THREAD

Главный поток живёт ровно столько же, сколько и сам процесс. Он не завершает свою работу раньше, потому что находится в состоянии зацикленности.

Фрагмент класса `ActivityThread`, где происходит уход в бесконечный цикл UI потока:

```
Looper.loop();
```

```
throw new RuntimeException("Main thread loop unexpectedly exited");
```

Если что-то пойдёт не так и выполнение программы продолжится, то будет выброшено исключение, однако этого не происходит.

LOOPER.LOOP

Внутри — бесконечный цикл обработки «событий»:

```
for (;;) {  
    Message msg = queue.next(); // might block  
    if (msg == null) {...}  
  
    // ... код пропущен ...
```

Q: что это всё значит?

A: это значит, что исполнение «тяжёлого» кода может привести к Freeze (замораживанию интерфейса), а как следствие к ANR — приложение не отвечает.



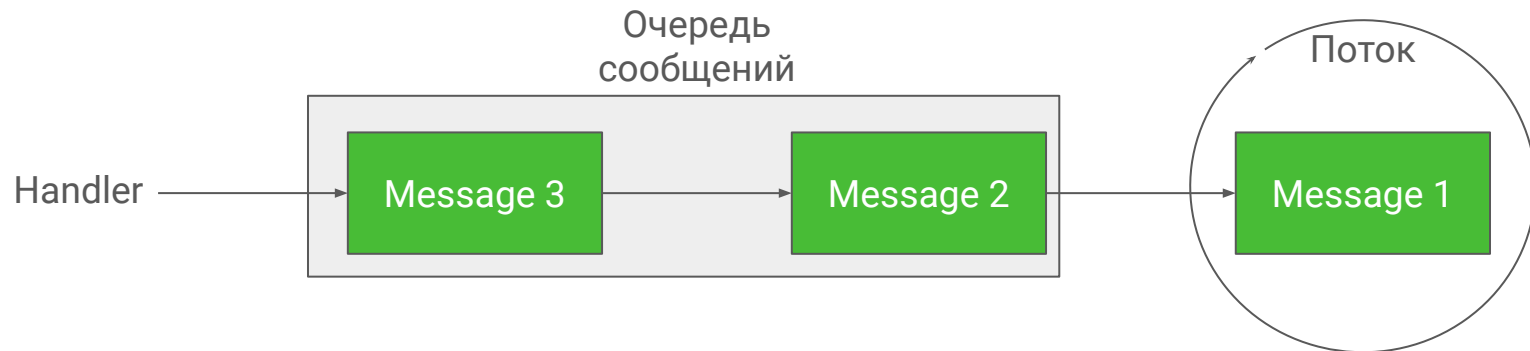
LOOPER

Стоит сразу уточнить, что это не совсем обычный бесконечный цикл.

Поток уходит в состояние ожидания до тех пор, пока какое-то событие не пробудит его. Событием может быть, к примеру, нажатие пользователя на экран.

MESSAGE QUEUE

У потока, для которого был вызван метод `loop`, существует очередь событий `MessageQueue`:



Сообщение представлено классом `Message`. Для передачи сообщения в очередь нужно воспользоваться классом `Handler`.

FREEZING

Давайте попробуем в onCreate запустить «тяжёлый» цикл, эмулирующий работу:

```
var sum = 0
for (i in 0..1_000_000_000_000) {
    sum++
}
println(sum)
```

Экран нашего приложения «заморозится» (freeze) и не будет отвечать на клики и другие события, поскольку UI поток будет занят вычислениями.



FREEZING

С точки зрения пользователя это будет выглядеть как «зависание» приложения.

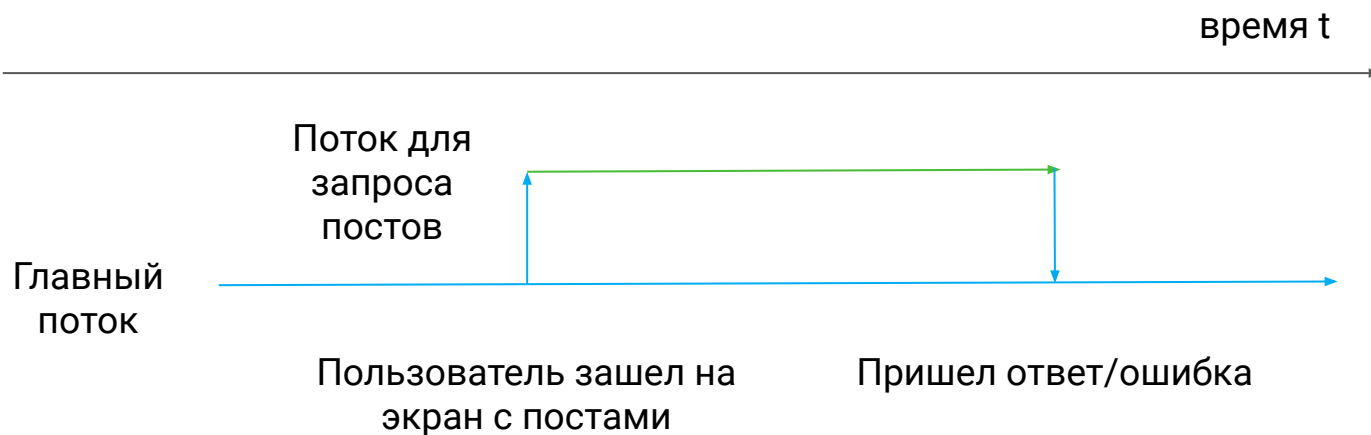
Точно так же с сетевыми запросами: в зависимости от скорости соединения запрос может занимать несколько секунд и пользователь будет думать, что приложение зависло.

Поэтому разработчики Android просто запретили делать запросы из основного потока приложения.

BACKGROUND THREAD

Q: Но если из основного потока нельзя, то как же делать запросы?

A: Можно создать другой поток, который часто называют background, и делать из него запросы. UI Thread по-прежнему будет отвечать за отрисовку экрана и обработку событий, а background будет ожидать завершения запроса (или окончания тяжёлых вычислений):



THREAD

В Java есть класс [Thread](#), который позволяет работать с потоками:

- ✓ C Thread
 - > E State
 - > I UncaughtExceptionHandler
 - m Thread()
 - m Thread(Runnable)
 - m Thread(ThreadGroup, Runnable)
 - m Thread(String)
 - m Thread(ThreadGroup, String)
 - m Thread(Runnable, String)
 - m Thread(ThreadGroup, Runnable, String)
 - m Thread(ThreadGroup, Runnable, String, long)
 - m blockedOn(Interruptible): void
 - m currentThread(): Thread
 - m yield(): void
 - m sleep(long): void
 - m sleep(long, int): void
 - m start(): void
 - m run(): void



УПРАВЛЕНИЕ СОСТОЯНИЕМ



STATE

Поскольку запрос в сеть может занимать продолжительное время, то чтобы пользователь не скучал, покажем ему на экране ProgressBar на время загрузки. В случае ошибки нужно дать возможность попробовать снова.



STATE

Для этого немного изменим интерфейс запроса постов. В ответ вернём описание текущего состояния экрана:

```
data class FeedModel(  
    val posts: List<Post> = emptyList(),  
    val loading: Boolean = false,  
    val error: Boolean = false,  
    val empty: Boolean = false,  
)
```

Подумайте, какие ещё состояния может содержать экран со списком постов?



STATE

Добавим во ViewModel немного динамики:

```
class PostViewModel(application: Application) : AndroidViewModel(application) {  
    // упрощённый вариант  
    private val repository: PostRepository = PostRepositoryImpl()  
    private val _data = MutableLiveData(FeedModel())  
    val data: LiveData<FeedModel>  
        get() = _data  
    val edited = MutableLiveData(empty)
```



STATE

Запрос списка постов вынесем на фоновый поток при помощи функции [thread](#), которая создаст и запустит в отдельном потоке указанный блок кода):

В отдельном потоке

```
fun loadPosts() {  
    thread {  
        // Начинаем загрузку  
        _data.postValue(FeedModel(loading = true))  
        try {  
            // Данные успешно получены  
            val posts = repository.getAll()  
            FeedModel(posts = posts, empty = posts.isEmpty())  
        } catch (e: IOException) {  
            // Получена ошибка  
            FeedModel(error = true)  
        }.also(_data::postValue)  
    }  
}
```



POSTVALUE

→ `_data.postValue(FeedModel(loading = true))`

Мы используем `postValue` для записи в LiveData с фонового потока, т.к. этот метод выполняет доставку данных в главный поток, в то время как обычный вызов `setValue` таких преобразований не делает.

Если попробовать пойти против этого правила, то получим Exception:

```
2020-11-25 01:16:35.332 2061-2129/ru.netology.nmedia E/AndroidRuntime: FATAL EXCEPTION: Thread-2
Process: ru.netology.nmedia, PID: 2061
java.lang.IllegalStateException: Cannot invoke setValue on a background thread
    at androidx.lifecycle.LiveData.assertMainThread(LiveData.java:462)
    at androidx.lifecycle.LiveData.setValue(LiveData.java:304)
    at androidx.lifecycle.MutableLiveData.setValue(MutableLiveData.java:50)
    at ru.netology.nmedia.viewmodel.PostViewModel$loadPosts$1.invoke(PostViewModel.kt:47)
    at ru.netology.nmedia.viewmodel.PostViewModel$loadPosts$1.invoke(PostViewModel.kt:21)
    at kotlin.concurrent.ThreadsKt$thread$thread$1.run(Thread.kt:30)
```

Также мы видим, что у нового потока свой собственный stack trace



FRAGMENT_FEED.XML

<TextView

```
    android:id="@+id/retryTitle"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginBottom="@dimen/common_spacing"
    android:text="@string/error_loading"
    app:layout_constraintBottom_toTopOf="@+id/retryButton"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintVertical_chainStyle="packed" />
```

<Button

```
    android:id="@+id/retryButton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/retry_loading"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@id/retryTitle" />
```

Данные 2 View для удобства можно объединить в [группу](#)



PROGRESSBAR

Рядом расположим состояние загрузки:

```
<ProgressBar
    android:id="@+id/progress"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:visibility="gone"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```




MATERIAL DESIGN

Если данных нет, то ни в коем случае не оставляем экран пустым.

Поскольку мы следуем [гайдлайнам](#) Material дизайна.

```
<TextView
    android:id="@+id/emptyText"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/empty_posts"
    android:visibility="gone"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```



FEEDFRAGMENT

`FeedFragment` теперь подписывается не только на посты, но и на все остальные состояния:

```
viewModel.data.observe(viewLifecycleOwner, { state ->
    adapter.submitList(state.posts)
    binding.progress.isVisible = state.loading
    binding.errorGroup.isVisible = state.error
    binding.emptyText.isVisible = state.empty
})

binding.retryButton.setOnClickListener { it: View!
    viewModel.loadPosts()
}
```



SINGLE LIVE EVENT



SINGLE LIVE EVENT

Некоторые события не должны менять стэйт приложения и должны выполняться всего один раз.

К примеру, всплывающее сообщение не должно быть показано пользователю дважды.

Для оповещения о таких операциях мы можем создать свой вид LiveData (см. следующий слайд).



SINGLE LIVE EVENT

```
class SingleLiveEvent<T> : MutableLiveData<T>() {  
    // FIXME: упрощённый вариант, пока не прошли Atomic'u  
    private var pending = false  
  
    override fun observe(owner: LifecycleOwner, observer: Observer<in T?>) {  
        require (!hasActiveObservers()) {  
            error("Multiple observers registered but only one will be notified of changes.")  
        }  
  
        super.observe(owner) { it: T!  
            if (pending) {  
                pending = false  
                observer.onChange(it)  
            }  
        }  
    }  
  
    override fun setValue(t: T?) {  
        pending = true  
        super.setValue(t)  
    }  
}
```



SINGLE LIVE EVENT

Поскольку в нашем случае навигация не является частью состояния бизнес логики, а управляется фреймворком навигации, то необходимо единовременно (т.е. всего один раз) оповещать фрагмент о создании поста:

```
private val _postCreated = SingleLiveEvent<Unit>()
val postCreated: LiveData<Unit>
    get() = _postCreated
```



SINGLE LIVE EVENT

Запрос создания поста:

```
fun save() {  
    edited.value?.let { it: Post  
        thread {  
            repository.save(it)  
            _postCreated.postValue(Unit)  
        }  
    }  
    edited.value = empty  
}
```



SINGLE LIVE EVENT

Со стороны `NewPostFragment` практически ничего не меняется:

```
binding.ok.setOnClickListener { it: View!  
    viewModel.changeContent(binding.edit.text.toString())  
    viewModel.save()  
    AndroidUtils.hideKeyboard(requireView())  
}  
viewModel.postCreated.observe(viewLifecycleOwner) {  
    viewModel.loadPosts()  
    findNavController().navigateUp()  
}
```




VIEWMODEL

Все остальные вызовы функций репозитория:

```
fun likeById(id: Long) {
    thread { repository.likeById(id) }
}

fun removeById(id: Long) {
    thread {
        // Оптимистичная модель
        val old = _data.value?.posts.orEmpty()
        _data.postValue(
            _data.value?.copy(posts = _data.value?.posts.orEmpty()
                .filter { it.id != id }
            )
        )
        try {
            repository.removeById(id)
        } catch (e: IOException) {
            _data.postValue(_data.value?.copy(posts = old))
        }
    }
}
```



ИТОГИ



ИТОГИ

Сегодня мы интегрировали наше мобильное приложение с сервером.

Мы использовали самый простейший подход с созданием потоков на каждый запрос.