План занятия

1. Задача
2. Retrofit
3. Настройка
4. Использование
5. Дополнительные настройки
6. Итоги
7. Домашнее задание

ЗАДАЧА

ЗАДАЧА

На прошлых лекциях мы разобрали, какие инструменты используются для работы с многопоточностью в Java и Android. А также посмотрели на ключевые библиотеки для работы:

- OkHttp — HTTP-запросы
- Glide — загрузка изображений

Сегодня наша задача: интеграция библиотеки Retrofit, которая позволит в декларативном стиле работать с HTTP-запросами (примерно так же, как мы в декларативном стиле работали с SQL-запросами в Room).

RETROFIT

RETROFIT

Ещё одна библиотека от Square (наряду с OkHttp, Picasso), позволяющая в декларативном стиле описывать взаимодействие с веб-сервисом:

```
public interface GitHubService {
  @GET("users/{user}/repos")
  Call<List<Repo>> listRepos(@Path("user") String user);
}
```

Таким образом, мы можем обеспечить RPC (Remote Procedure Call) по HTTP, избегая необходимости вручную парсить JSON и строить URL'ы запроса (хорошо работает при REST like API).

RETROFIT

```
public interface GitHubService {
  @GET("users/{user}/repos")
  Call<List<Repo>> listRepos(@Path("user") String user);
}
```

```
Call<List<Repo>> repos = service.listRepos("octocat");
```

ПОДКЛЮЧЕНИЕ

```
implementation "com.squareup.retrofit2:retrofit:$retrofit_version"
implementation "com.squareup.retrofit2:converter-gson:$retrofitgson_version"
```

Обратите внимание: Retrofit «тянет за собой» OkHttp, поэтому явное указание зависимости от OkHttp в build.gradle можно убрать (Retrofit подключит необходимую ему версию OkHttp сам).

ПОДКЛЮЧЕНИЕ

Кроме того, Retrofit умеет интегрироваться с Gson для того, чтобы организовать парсинг JSON-ответов. Поэтому можно убрать прямую зависимость от Gson (поскольку он автоматически подтянется за счёт converter-gson):

```
implementation "com.squareup.retrofit2:retrofit:$retrofit_version"
implementation "com.squareup.retrofit2:converter-gson:$retrofitgson_version"
```

НАСТРОЙКА

НАСТРОЙКА

Для настройки нам нужно:

1. задать BASE_URL
2. создать экземпляр Retrofit с BASE_URL и Gson Converter'ом:

```
private const val BASE_URL = "http://10.0.2.2:9999/api/slow/"

private val retrofit = Retrofit.Builder()
    .addConverterFactory(GsonConverterFactory.create())
    .baseUrl(BASE_URL)
    .build()
```

CONVERTER

Всё, что делает Converter — преобразует тело запроса/ответа в нужный формат:

```
JsonReader jsonReader = gson.newJsonReader(value.charStream());
try {
  T result = adapter.read(jsonReader);
  if (jsonReader.peek() != JsonToken.END_DOCUMENT) {
    throw new JsonIOException("JSON document was not fully consumed.");
  }
  return result;
} finally {
  value.close();
}
```

Пример преобразования JSON'а в тип T, реализованный в GsonConverter.

ОСНОВНАЯ ИДЕЯ

Дальше всё достаточно просто: мы описываем интерфейсы (как и с Room) и помечаем методы аннотациями:

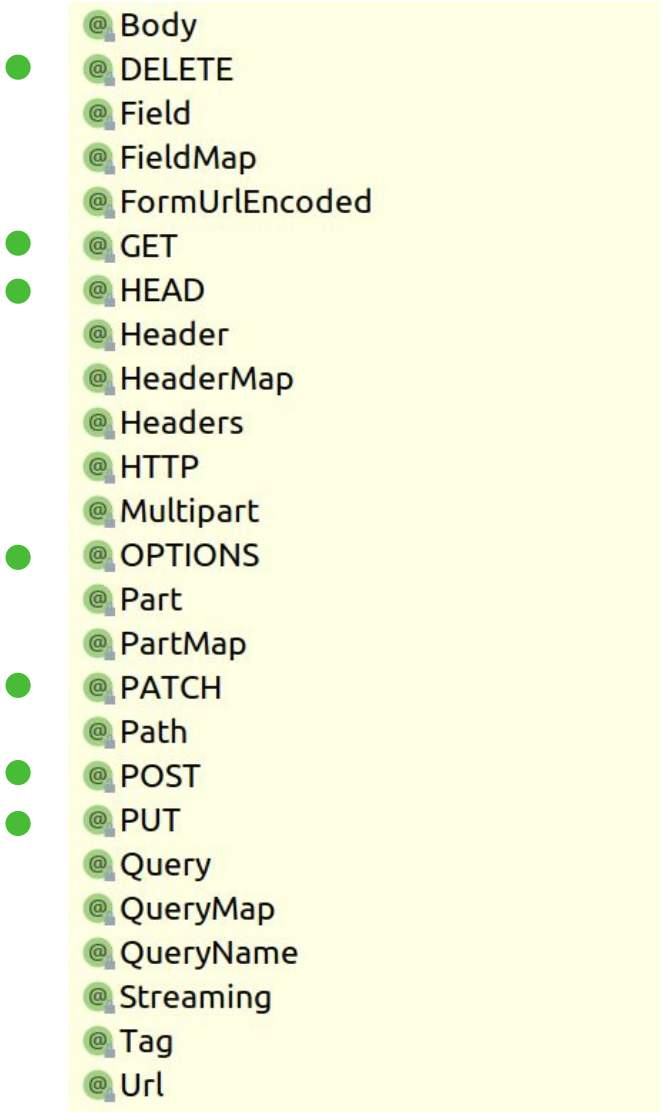
```
interface PostsApiService {
    @GET( value: "posts")
    fun getAll(): Call<List<Post>>
}
```

Начнём с Call, а затем вернёмся к аннотациям.

Call — это интерфейс, позволяющий выполнить запрос (sync/async):

```java
public interface Call<T> extends Cloneable {
  /** Synchronously send the request and return its response. ...*/
  Response<T> execute() throws IOException;

  /** Asynchronously send the request and notify {@code callback} of its response or if an error ...*/
  void enqueue(Callback<T> callback);

  /** Returns true if this call has been either {@linkplain #execute() executed} or {@linkplain ...*/
  boolean isExecuted();

  /** Cancel this call. An attempt will be made to cancel in-flight calls, and if the call has not ...*/
  void cancel();

  /** True if {@link #cancel()} was called. */
  boolean isCanceled();

  /** Create a new, identical call to this one which can be enqueued or executed even if this call ...*/
  Call<T> clone();

  /** The original HTTP request. */
  Request request();

  /** Returns a timeout that spans the entire call: resolving DNS, connecting, writing the request ...*/
  Timeout timeout();
}
```

АННОТАЦИИ



Аннотации, обозначающие методы HTTP

- GET — получение ресурса (не имеет тела)
- POST — создание/обновление ресурса
- PUT — обновление ресурса
- PATCH — обновление ресурса (чаще всего — частичное)
- DELETE — удаление ресурса
- HEAD, OPTIONS — дополнительные

АННОТАЦИИ

Все аннотации HTTP-методов внутри устроены одинаково:

```
@Documented
@Target(METHOD)
@Retention(RUNTIME)
public @interface GET {
```

> A relative or absolute path, or full URL of the endpoint. This value is optional if the first parameter of the method is annotated with `@Url`.
>
> See base URL for details of how this is resolved against a base URL to create the full endpoint URL.

```
  String value() default "";
}
```

АННОТАЦИИ

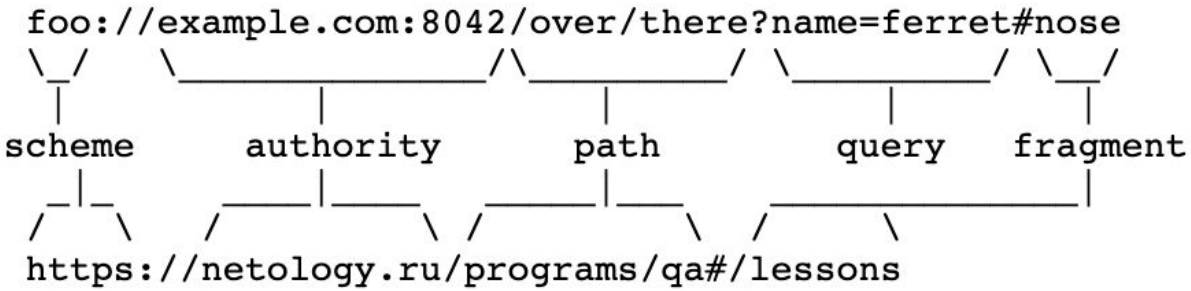
Напоминаем:

- если в аннотации элемент называется value и вы используете только этот элемент, то можно вместо @GET(value = "...") писать просто @GET("...")
- если у value есть default значение, то можно не писать и круглые скобки, т.е. вместо @GET("") можно @GET

URI

Перед тем как обсудить особенность построения URL запроса, вспомним схему URI:

```
URI         = scheme ":" hier-part [ "?" query ] [ "#" fragment ]
```



Напомним, URL — частный случай URI.

BASE_URL

```
BASE_URL = "http://10.0.2.2/api/slow/"
```

У Retrofit есть несколько правил для работы с BASE_URL:

1. BASE_URL должен оканчиваться /
2. Если не писать в аннотациях метода начальный /, то адрес прибавляется к BASE_URL:
   @GET("posts") = http://10.0.2.2/api/slow/posts
3. Если писать начальный слэш, то замещает весь путь:
   @GET("/posts") = http://10.0.2.2/posts
4. Если писать полный адрес без схемы, то схема остаётся, адрес меняется:
   @GET("//nmedia.dev/api/posts") = http://nmedia.dev/api/posts
5. Если писать полный адрес со схемой, то всё заменяется:
   @GET("https://nmedia.dev/api/posts") = https://nmedia.dev/api/posts

SINGLETON

Для доступа к API создаётся Singleton с lazy-инициализацией поля:

```
object PostsApi {
    val retrofitService : PostsApiService by lazy {
        retrofit.create(PostsApiService::class.java)
    }
}
```

SINGLETON

object Singleton создаётся как статичное поле в момент срабатывания инициализатора класса. Напомним, «за сценой» Kotlin генерирует нечто подобное:

```
public final class PostsApi {
  public final static PostsApi INSTANCE = new PostsApi();

  private PostsApi() {}
}
```

А все вызовы вида PostsApi. превращаются в PostsApi.INSTANCE.

BY LAZY

by lazy — это потокобезопасный делегат ленивой инициализации:

```
public actual fun <T> lazy(initializer: () -> T): Lazy<T> = SynchronizedLazyImpl(initializer)
```

А внутри уже знакомая нам конструкция с volatile и synchronized (см. следующий слайд).

```
private var initializer: (() -> T)? = initializer
@Volatile private var _value: Any? = UNINITIALIZED_VALUE
// final field is required to enable safe publication of constructed instance
private val lock = lock ?: this

override val value: T
    get() {
        val _v1 = _value
        if (_v1 !== UNINITIALIZED_VALUE) {
            @Suppress( ...names: "UNCHECKED_CAST")
            return _v1 as T
        }

        return synchronized(lock) {
            val _v2 = _value
            if (_v2 !== UNINITIALIZED_VALUE) {
                @Suppress( ...names: "UNCHECKED_CAST") (_v2 as T)
            } else {
                val typedValue = initializer!!()
                _value = typedValue
                initializer = null
                typedValue  ^synchronized
            }
        }
    }
```

PROXY

Теперь самое интересное: как получается так, что у интерфейсов появляются реализации?

```
retrofit.create(PostsApiService::class.java)
```

? *Вопрос: помните ли вы, как для интерфейсов DAO появлялась реализация?*

Демонстрация в debugger'e

PROXY

Здесь используется другая реализация: на основе Proxy из состава JDK:

```
Proxy.newProxyInstance(
    service.getClassLoader(),
    new Class<?>[] {service},
    new InvocationHandler() {
      private final Platform platform = Platform.get();
      private final Object[] emptyArgs = new Object[0];

      @Override
      public @Nullable Object invoke(Object proxy, Method method, @Nullable Object[] args)
          throws Throwable {...}
    });
```

В чём суть: при запуске приложения создаётся специальный объект, все вызовы методов которого (в нашем случае getAll) попадают в специальный обработчик (InvocationHandler).

```
@Override
public @Nullable Object invoke(Object proxy, Method method, @Nullable Object[] args)  proxy:
    throws Throwable {
  // If the method is a method from Object then defer to normal invocation.
  if (method.getDeclaringClass() == Object.class) {...}
  args = args != null ? args : emptyArgs;  args: null  emptyArgs: Object[0]@4702
  return platform.isDefaultMethod(method)
      ? platform.invokeDefaultMethod(method, service, proxy, args)
      : loadServiceMethod(method).invoke(args);
```

Variables

- this = {Retrofit$1@4699}
- service = {Class@4694} "interface ru.netology.nmedia.api.PostsApiService" ... Navigate
- proxy = {$Proxy1@4700} "retrofit2.Retrofit$1@310f94ef"
- method = {Method@4701} "public abstract retrofit2.Call ru.netology.nmedia.api.PostsApiService.getAll()"
- args = null
- emptyArgs = {Object[0]@4702}
- platform = {Platform$Android@4703}

T.e. Retrofit перехватывает все вызовы, анализирует аннотации, проставленные над методом, и выполняет запрос.

ИСПОЛЬЗОВАНИЕ

ИСПОЛЬЗОВАНИЕ

Теперь, когда у нас есть глобальный синглтон, мы можем его использовать так же, как использовали Glide/Picasso* (только им требовался контекст, а нашему объекту — нет).

Примечание*: конечно вы должны понимать, что реализация нашего синглтона немного отличается, но суть одна и та же.

EXECUTE/ENQUEUE

Так же, как это было с OkHttp, нам предоставляется два метода исполнения запросов:

- execute (синхронный)
- enqueue (асинхронный) с Callback'ом, который в Android будет вызван в UI Thread'е* (см. следующий слайд)

Примечание*: надеемся, что вы ещё не забыли про Handler, Looper и MainLooper.

EXECUTE/ENQUEUE

```
static final class Android extends Platform {
  Android() { super( hasJava8Types: Build.VERSION.SDK_INT >= 24); }

  @Override
  public Executor defaultCallbackExecutor() { return new MainThreadExecutor(); }

  @Nullable
  @Override
  Object invokeDefaultMethod(
      Method method, Class<?> declaringClass, Object object, Object... args) throws Throwable {...}

  static final class MainThreadExecutor implements Executor {
    private final Handler handler = new Handler(Looper.getMainLooper());

    @Override
    public void execute(Runnable r) { handler.post(r); }
  }
}
```

CALLBACK

Поскольку мы будем использовать enqueue, то нам нужно использовать объект, удовлетворяющий интерфейсу Callback:

```
public interface Callback<T> {
    Invoked for a received HTTP response.
    Note: An HTTP response may still indicate an application-level failure
    such as a 404 or 500. Call Response.isSuccessful() to
    determine if the response indicates success.
  void onResponse(Call<T> call, Response<T> response);

    Invoked when a network exception occurred talking to the server or
    when an unexpected exception occurred creating the request or
    processing the response.
  void onFailure(Call<T> call, Throwable t);
}
```

CALLBACK

Обратите внимание: onResponse вызывается и для не 2xx кодов ответов*:

```
public interface Callback<T> {
    Invoked for a received HTTP response.
    Note: An HTTP response may still indicate an application-level failure
    such as a 404 or 500. Call Response.isSuccessful() to
    determine if the response indicates success.
  void onResponse(Call<T> call, Response<T> response);

    Invoked when a network exception occurred talking to the server or
    when an unexpected exception occurred creating the request or
    processing the response.
  void onFailure(Call<T> call, Throwable t);
}
```

Примечание*: во многих примерах забывают рассказать об этом.

HTTP

Статус-коды ответов:

- 1xx — информационные
- 2xx — успешно
- 3xx — перенаправление
- 4xx — ошибки клиента
- 5xx — ошибки сервера

ONRESPONSE

Ключевые ошибки:

1. Не обрабатывают не 2xx статусы
2. Считают, что успешный статус это 200 (OK)

```kotlin
class PostRepositoryImpl : PostRepository {
    override fun getAllAsync(callback: PostRepository.GetAllCallback) {
        PostsApi.retrofitService.getAll().enqueue(object : Callback<List<Post>> {
            override fun onResponse(call: Call<List<Post>>, response: Response<List<Post>>) {
                // response.isSuccessful - 2xx код ответа
                if (!response.isSuccessful) {
                    // response.message() - статус сообщение ответа
                    // response.code() - статус код ответа
                    // response.errorBody() - raw-body ответа
                    // для примера:
                    callback.onError(RuntimeException(response.message()))
                    return
                }

                // response.headers() - заголовки ответа
                // response.raw() - необработанное тело ответа
                // response.body() - приведённое с помощью конвертера к типу List<Post>?
                callback.onSuccess( posts: response.body() ?: throw RuntimeException("body is null"))
            }

            override fun onFailure(call: Call<List<Post>>, t: Throwable) {
                TODO( reason: "Not yet implemented")
            }
        })
    }
```

PATH PARAMS

```
@GET( value: "posts/{id}")
fun getById(@Path( value: "id") id: Long): Call<Post>

@DELETE( value: "posts/{id}")
fun removeById(@Path( value: "id") id: Long): Call<Unit>

@POST( value: "posts/{id}/likes")
fun likeById(@Path( value: "id") id: Long): Call<Post>

@DELETE( value: "posts/{id}/likes")
fun dislikeById(@Path( value: "id") id: Long): Call<Post>
```

Placeholder'ы в аннотациях методов запроса {} в совокупности с аннотацией @Path используются для подстановки параметров в путь запроса.

likeById(1) = POST "${BASE_URL}/posts/**1**/likes"

@PATH

```java
@Documented
@Retention(RUNTIME)
@Target(PARAMETER)
public @interface Path {
  String value();

  boolean encoded() default false;
}
```

> Specifies whether the argument value to the annotated method parameter is already URL encoded.

URL Encoding (так же и Percent Encoding) — специальная кодировка, преобразующая URL-небезопасные символы в URL'е (например, русский язык, пробелы и т.д.)

CALL<UNIT>

```
@DELETE( value: "posts/{id}")
fun removeById(@Path( value: "id") id: Long): Call<Unit>
```

Отдельно стоит отметить, что если мы не ждём ответа от сервиса, то стоит указывать именно Call<Unit> (тогда сработает Converter, который закроет тело ответа):

```
static final class UnitResponseBodyConverter implements Converter<ResponseBody, Unit> {
  static final UnitResponseBodyConverter INSTANCE = new UnitResponseBodyConverter();

  @Override
  public Unit convert(ResponseBody value) {
    value.close();
    return Unit.INSTANCE;
  }
}
```

BODY

Для отправки тела запроса есть маркерная аннотация (значит не имеющая элементов) @Body:

```
@POST( value: "posts")
fun save(@Body post: Post): Call<Post>
```

> Use this annotation on a service method param when you want to directly control the request body of a POST/PUT request (instead of sending in as request parameters or form-style request body). The object will be serialized using the `Retrofit` instance `Converter` and the result will be set directly as the request body.
>
> Body parameters may not be `null`.

```
@Documented
@Target(PARAMETER)
@Retention(RUNTIME)
public @interface Body {}
```

ДОПОЛНИТЕЛЬНЫЕ НАСТРОЙКИ

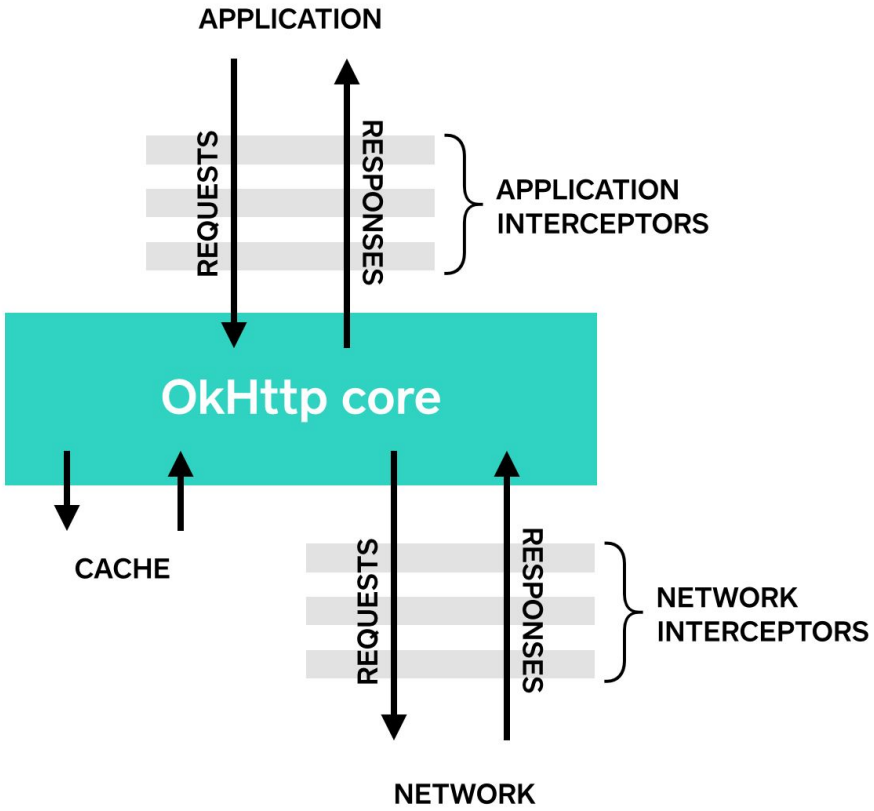
ДОПОЛНИТЕЛЬНЫЕ НАСТРОЙКИ

Две самые частые настройки, которые нужны при использовании Retrofit:

1. Логирование запросов
2. Смена URL'а в зависимости от типа сборки (Debug/Release)

ЛОГИРОВАНИЕ

Логирование основано на идее применения interceptor'ов (перехватчиков запросов):



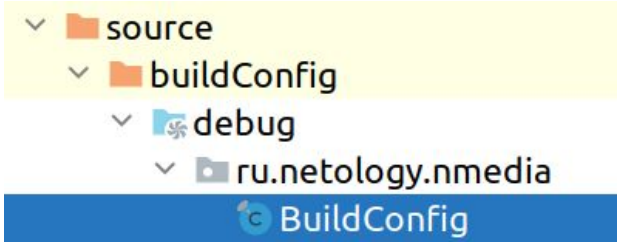
Изображение с сайта okhttp

ЛОГИРОВАНИЕ

```
implementation "com.squareup.okhttp3:logging-interceptor:$okhttplogging_version"

private val logging = HttpLoggingInterceptor().apply {  this: HttpLoggingInterceptor
    if (BuildConfig.DEBUG) {
        level = HttpLoggingInterceptor.Level.BODY
    }
}

private val okhttp = OkHttpClient.Builder()
    .addInterceptor(logging)
    .build()

private val retrofit = Retrofit.Builder()
    .addConverterFactory(GsonConverterFactory.create())
    .baseUrl(BASE_URL)
    .client(okhttp)
    .build()
```

BUILDCONFIG

BuildConfig — класс, генерируемый при сборке на основании данных из build.gradle:




```
public final class BuildConfig {
  public static final boolean DEBUG = Boolean.parseBoolean( s: "true");
  public static final String APPLICATION_ID = "ru.netology.nmedia";
  public static final String BUILD_TYPE = "debug";
  public static final int VERSION_CODE = 1;
  public static final String VERSION_NAME = "1.0";
}
```

BUILDCONFIG

Именно на основании его полей, мы включаем определённый уровень логирования (по умолчанию стоит None):

```
@set:JvmName( name: "level")
@Volatile var level = Level.NONE

enum class Level {
  /** No logs. */
  NONE,
```

BUILDCONFIG

Мы также можем добавлять в BuildConfig свои поля с помощью Gradle:

```
buildTypes {
    release {
        minifyEnabled false
        proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'
        manifestPlaceholders.usesCleartextTraffic = false
        buildConfigField "String", "BASE_URL", '"https://nmedia.herokuapp.com"'
    }
    debug {
        manifestPlaceholders.usesCleartextTraffic = true
        buildConfigField "String", "BASE_URL", '"http://10.0.2.2:9999"'
    }
}
```

BUILDCONFIG

```
private const val BASE_URL = "${BuildConfig.BASE_URL}/api/slow"
```

Для генерации новой версии файла необходимо вызвать Make Project (Ctrl + F9).

ИТОГИ

ИТОГИ

Сегодня мы посмотрели на базовую настройку и использование Retrofit, а также заглянули «под капот», чтобы понять, как всё устроено (ничего нового, кроме Proxy там нет).

Мы изучили не все аннотации, но по мере развития нашего приложения (добавления поиска, загрузки файлов, аутентификации) рассмотрим и оставшиеся.

ИТОГИ

Ключевое для нас: несмотря на то, что Retrofit немного упростил нам жизнь (но усложнил настройку), самое неудобное — это Callback'и.

Поэтому со следующей лекции мы с вами будем знакомиться с корутинами, которые обеспечат нас возможностью «писать асинхронный код так, как будто мы пишем синхронный»*.

Примечание*: конечно же, это достаточно утрированное сравнение.