**Лекция 4. Представления, основанные на данных**

[В предыдущей лекции](https://info448.github.io/resources-and-layouts.html#resources-and-layouts) обсуждалось, как использовать представления для отображения контента и поддержки взаимодействия с пользователем. Эта лекция расширяет эти концепции и представляет методы создания **представлений, управляемых данными** — представлений, которые могут *динамически* представлять модель данных в виде *прокручиваемого списка* . В ней также объясняется, как получить доступ к данным в Интернете с помощью библиотеки [Volley](https://developer.android.com/training/volley/index.html) . В целом, этот процесс демонстрирует распространенный способ соединения пользовательского интерфейса для приложения (определенного как XML) с логикой и элементами управления данными (определенными в Java), следуя архитектуре **Model-View-Controller,** встречающейся во всей структуре Android.

В этой лекции используется код, найденный по адресу <https://github.com/info448/lecture04-lists> .

**4.1 ListView и адаптеры**

В частности, в этой лекции обсуждалось, как использовать [ListView](https://developer.android.com/guide/topics/ui/layout/listview.html)[12](https://info448.github.io/data-views.html#fn12) , который представляет собой ViewGroup, отображающий прокручиваемый список элементов! A ListViewпо сути является LinearLayoutвнутренним элементом ScrollView(который представляет собой , ViewGroupкоторый можно прокручивать). Каждый элемент внутри LinearLayout является другим View(обычно Layout), представляющим определенный элемент в списке.

Но ListViewделает дополнительную работу, выходящую за рамки простого вложения Views: он отслеживает, какие элементы уже отображаются на экране, раздувая только видимые элементы (плюс несколько дополнительных сверху и снизу в качестве буферов). Затем, когда пользователь прокручивает, ListView берет исчезающие представления и *перерабатывает* их (изменяя их содержимое, но не перераздувая с нуля), чтобы повторно использовать их для новых появляющихся элементов. Это позволяет ему экономить память, обеспечивать лучшую производительность и в целом работать более плавно. Смотрите [это руководство](https://github.com/codepath/android_guides/wiki/Using-an-ArrayAdapter-with-ListView#row-view-recycling) для диаграмм и дальнейшего объяснения этого поведения переработки.

* Обратите внимание, что более продвинутая и гибкая версия этого поведения предлагается классом [RecyclerView](https://developer.android.com/guide/topics/ui/layout/recyclerview.html), который работает в основном так же, но требует нескольких дополнительных шагов для настройки. См. также [это руководство](https://developer.android.com/training/material/lists-cards.html) для получения более подробной информации.

Элемент управления ListViewиспользует архитектуру [**Model-View-Controller (MVC)**](https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller) . Это шаблон проектирования, общий для систем пользовательского интерфейса, который организует программы в три части:

1. Модель , представляющая собой данные или информацию в системе **.**
2. Представление , которое является отображением или представлением этих данных **.**
3. Контроллер , который выступает в качестве посредника между Моделью и Представлением и связывает их вместе **.**

Шаблон MVC можно найти повсюду в Android. На высоком уровне ресурсы предоставляют *модели* и *представления* (отдельно), в то время как Java Activities действуют как *контроллеры* .

*Интересный факт* : шаблон «Модель-Представление-Контроллер» изначально был разработан как часть языка [Smalltalk](http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html) , который был первым объектно-ориентированным языком!

Таким образом, чтобы использовать ListView, у нас будут некоторые данные для отображения ( **модель** ), **представления** (макеты), которые будут показаны, и ListViewсам будет связывать их вместе, выступая в качестве **контроллера** . В частности, ListViewявляется подклассом [AdapterView](https://developer.android.com/reference/android/widget/AdapterView.html), который является представлением, поддерживаемым источником данных — AdapterViewсуществует для связывания представления и данных вместе (как и должен делать контроллер).

* Есть AdapterViewsи другие. Например, [GridView](https://developer.android.com/guide/topics/ui/layout/gridview.html)работает точно так же, как ListView, но размещает элементы в прокручиваемой сетке, а не в прокручиваемом списке.

Чтобы использовать ListView, нам нужно расставить все по местам:

1. Сначала мы указываем **модель** : некоторые необработанные данные. Начнем с простого списка строк, заполняя его данными-заполнителями:
2. **val** data = mutableListOf<**String**>()
3. **for**(i **in** 99 downTo 1){
4. **data**.add(i.toString() + " bottles of beer on the wall")

}

Хотя обычно мы должны определять такие жестко закодированные данные как [ресурс XML](https://developer.android.com/guide/topics/resources/string-resource.html#StringArray) , мы создадим его динамически для тестирования (и чтобы сделать его изменяемым позже!). Использование списка вместо массива упрощает построение (и работает так же).

1. Далее мы указываем **вид** : a, Viewкоторый будет отображаться для каждого элемента данных в списке. Определим ресурс макета XML для этого ( list\_itemэто хорошее название и общепринятая идиома).

Нам на самом деле не нужно указывать полный макет (хотя мы могли бы, если бы захотели): TextViewдостаточно просто базового. Укажите ширину match\_parentи высоту wrap\_content. *Не забудьте id!*

*<!-- need to include the XML namespace (xmlns) so the `android` namespace validates -->*

**<TextView** xmlns:android="http://schemas.android.com/apk/res/android"

android:id="@+id/txtItem"

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content" **/>**

Чтобы он выглядел лучше, вы можете указать android:minHeight="?android:attr/listPreferredItemHeight"(используя предпочтительную высоту фреймворка для списков), и немного center\_verticalгравитации. android:linesСвойство также полезно, если вам нужно больше места.

1. Наконец, мы указываем **контроллер** : ListViewсам. Добавьте этот элемент в ресурс макета Activity ( *практика* : какими должны быть его размеры?)

Чтобы завершить контроллер ListView, нам нужно предоставить ему [13](https://info448.github.io/data-views.html#fn13) , который соединит *модель* с *представлением* . Адаптер выполняет работу по «переводу» между моделью и представлением, выполняя сопоставление из типов данных (например, a ) и типов представлений (например, a ).[Adapter](https://developer.android.com/guide/topics/ui/declaring-layout.html#AdapterViews)StringTextView

В частности, мы будем использовать [ArrayAdapter](https://developer.android.com/reference/android/widget/ArrayAdapter.html), который является одним из самых простых адаптеров в использовании (и потому что у нас есть массив данных!). An ArrayAdapterсоздает представления, вызывая .toString()каждый элемент в массиве и устанавливая его Stringв качестве содержимого TextView!

*//kotlin*

**val** adapter = ArrayAdapter<**String**>(**this**,

R.layout.list\_item\_layout, R.layout.list\_item\_txtView, myStringArray);

* Обратите внимание на параметры конструктора: a Contextдля доступа к ресурсам, ресурс макета для использования для каждого элемента, TextViewвнутри этого макета (цель сопоставления) и массив данных (источник сопоставления). Также обратите внимание, что этот экземпляр использует [*обобщения*](https://info448.github.io/java-review.html#generics) : мы используем массив Strings(в отличие от массива Dogsили какого-либо другого типа).

Мы получаем ссылку на ListViewwith findViewById()и вызываем ListView#setAdapter()подключение адаптера к этому контроллеру.

*//kotlin*

**val** listView = findViewById<ListView>(R.id.list\_view)

listView.setAdapter(adapter)

И это все, что нужно для создания прокручиваемого списка данных! Чтобы отслеживать процесс: Адаптер пройдет по каждому элементу в *модели* и «переведет» этот элемент в содержимое Представления. Затем эти Представления будут отображаться в прокручиваемом списке.

Каждый элемент в этом списке можно выбрать (можно задать onClickобратный вызов). Это позволяет нам нажать на любой элемент, чтобы (например) получить более подробную информацию о нем. Используйте AdapterView#setOnItemClickListener(OnItemClickListener)функцию для регистрации обратного вызова.

* Параметр positionв onItemClick()обратном вызове — это индекс элемента, который был нажат. Используйте (Type)parent.getItemAtPosition(position)для доступа к значению данных, связанному с этим представлением.

Кроме того, каждый элемент имеет индивидуальный макет, поэтому вы можете настроить эти внешние виды (например, если наш макет также хочет включать изображения). Посмотрите [это руководство](https://github.com/codepath/android_guides/wiki/Using-an-ArrayAdapter-with-ListView#row-view-recycling) для примера создания пользовательского адаптера для заполнения нескольких элементов Viewsданными из списка!

И помните, a GridView— это по сути то же самое (на самом деле, мы можем просто изменить это и все заработает, если используем *полиморфизм* !) Обратите внимание, что тип данных для AdapterViewнемного запутанный, если вы хотите быть универсальным:

**val** adapterView = findViewById<AdapterView<ArrayAdapter<**String**>>>(R.id.list\_view)

**4.2 Сетевое взаимодействие с Volley**

Список с жестко закодированными данными не очень полезен. Было бы лучше, если бы к этим данным можно было получить динамический доступ, например, загрузить их из Интернета!

Существует несколько различных способов программной отправки сетевых запросов из приложения Android. «Самый низкий уровень» — использование [HttpURLConnection](https://developer.android.com/reference/javax/net/ssl/HttpsURLConnection.html)API. С помощью этого API вы вызываете методы для открытия соединения с URL-адресом, а затем отправляете HTTP-запрос в это место. Ответ возвращается как [InputStream](https://developer.android.com/reference/java/io/InputStream.html), который вам нужно «прочитать» байт за байтом, чтобы восстановить полученные данные (например, чтобы превратить их обратно в строку). Подробности см. [в этом примере .](https://developer.android.com/training/basics/network-ops/connecting.html#download)

Хотя этот метод эффективен, его реализация может быть утомительной. Более того, загрузка сетевых данных может занять некоторое время, а эти вызовы сетевых методов являются синхронными и **блокирующими** , поэтому не позволят другому коду выполняться во время загрузки, включая код, который включает пользовательский интерфейс! Такая блокировка приведет к печально известной ошибке *«Приложение не отвечает» (ANR)* . Хотя можно отправлять такие запросы асинхронно в *фоновом потоке,* чтобы избежать блокировки, это требует дополнительных накладных расходов. Подробнее см. [в лекции по службам .](https://info448.github.io/services.html#services)

Чтобы решить эти проблемы с меньшими усилиями, может быть более эффективным использовать **внешнюю библиотеку** , которая позволяет нам абстрагироваться от этого процесса и просто говорить о выполнении сетевых запросов и получении данных из них. (Это похоже на то, как в веб-программировании [fetchAPI](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API) абстрагирует непрозрачный XMLHttpRequestобъект). В частности, в этой лекции будет представлена ​​библиотека [**Volley**](https://developer.android.com/training/volley/index.html) , которая является внешней библиотекой, разработанной и поддерживаемой Google. Она обеспечивает ряд преимуществ по сравнению с более «ручным» подходом, включая обработку нескольких одновременных запросов и включение кэширования загруженных данных. Она также заставляет сетевые запросы обрабатываться асинхронно в фоновом потоке без каких-либо дополнительных усилий!

Главным «конкурентом» Volley является библиотека [Retrofit](http://square.github.io/retrofit/) , созданная [Square](http://square.github.io/) . Хотя Retrofit обычно быстрее обрабатывает загруженные данные, Volley имеет встроенную поддержку обработки изображений (которая будет полезна в будущем) и имеет немного более простой интерфейс.

**Использование залпа**

Поскольку Volley — это внешняя библиотека (она не встроена в фреймворк Android), вам нужно явно загрузить и включить ее в свой проект. К счастью, мы можем использовать систему сборки *Gradle* , чтобы сделать это за нас, указав Volley как **зависимость** для проекта. Внутри файла *уровня приложения* build.gradle добавьте следующую строку в dependenciesсписок:

compile 'com.android.volley:volley:1.0.0'

Это сообщит Android, что он должен загрузить и включить версию 1.0.0библиотеки Volley при сборке приложения. Нажмите кнопку «Синхронизировать», чтобы обновить и перестроить проект.

* Внешние библиотеки будут встроены в ваше приложение, увеличивая размер скомпилированного файла .apk(больше кода!). Хотя это не вызовет никаких проблем для нас, это стоит иметь в виду при разработке новых приложений.

После включения Volley в качестве зависимости у вас появится доступ к [классам и API](http://afzaln.com/volley/) для использования в вашем коде.

Чтобы запросить данные с помощью Volley, вам необходимо создать экземпляр [Request](https://afzaln.com/volley/com/android/volley/Request.html)объекта на основе типа данных, которые вы будете загружать: StringRequestдля загрузки текстовых данных, JsonRequestдля загрузки данных в формате JSON или ImageRequestдля загрузки изображений.

StringRequestНапример, конструктор для принимает 4 аргумента:

1. Константа, представляющая HTTP-метод (глагол), который следует использовать. Например,Request.Method.GET
2. URL-адрес для отправки запроса (в виде строки)
3. Объект Response.Listener, определяющий функцию обратного вызова, которая будет выполнена при получении ответа.
4. Объект Response.ErrorListener, определяющий функцию обратного вызова, которая будет выполнена в случае ошибки.

Поскольку последние два объекта *-слушателя* обычно определяются с помощью анонимных классов, это может сделать конструктор Request более сложным, чем он есть на самом деле (хотя использование лямбда-выражений в Kotlin помогает):

*//java*

*//silly example: get 20 random dinosaur names*

String url = "http://dinoipsum.herokuapp.com/api/?format=text&words=20&paragraphs=1";

Request myRequest = **new** StringRequest(Request.Method.GET, url,

**new** Response.Listener<String>() {

**public** void onResponse(String response) {

Log.v(TAG, response);

}

}, **new** Response.ErrorListener() {

@Override

**public** void onErrorResponse(VolleyError error) {

Log.e(TAG, error.toString());

}

});

*//kotlin*

**val** request = StringRequest(Request.Method.GET, urlString,

*//callback for success*

Response.Listener { response ->

Log.v(TAG, response)

},

*//callback for failure*

Response.ErrorListener { error ->

Log.e(TAG, error.toString())

})

(Также обратите внимание, что Response.Listenerэто *универсальный* класс, в котором мы указываем формат, в котором ожидаем получить ответ. Это Stringдля StringRequest, но было бы, например, JSONObjectдля JsonObjectRequest). Kotlin выполняет часть этой работы за нас.

Чтобы на самом деле *отправить* этот запрос, вам нужен [RequestQueue](https://afzaln.com/volley/com/android/volley/RequestQueue.html), который действует как «диспетчер» и обрабатывает отправку запросов в фоновых потоках и иным образом управляет сетевыми операциями. Мы создаем диспетчер с параметрами по умолчанию (для сети и кэширования) с помощью Volley.newRequestQueue()метода factory:

*//kotlin*

**val** requestQueue: requestQueue = Volley.newRequestQueue(**this**.applicationContext);

Фабричный метод принимает контекст для управления кэшем; лучше всего использовать контекст приложения, чтобы он не зависел от одного действия.

После того, как у вас есть RequestQueue, вы можете добавить в него свой запрос, чтобы «отправить» его:

requestQueue.add(myRequest);

Если вы протестируете этот код, вы заметите, что он не работает! Программа завершится с ошибкой SecurityException.

В качестве [функции безопасности](https://developer.android.com/guide/topics/permissions/index.html) приложения Android по умолчанию имеют очень ограниченный доступ к общей операционной системе (например, для выполнения чего-либо, кроме отображения макета). Приложение не может использовать Интернет (что может израсходовать тарифные планы пользователей!) без явного разрешения пользователя. Это разрешение предоставляется пользователем во *время установки* .

Чтобы получить разрешение, приложение должно его запросить («Мама, можно мне…?»). Мы делаем это, заявляя, что приложение использует Интернет в файле AndroidManifest.xml(в котором есть все данные о нашем приложении!)

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

*<!-- put this ABOVE the <application> tag -->*

Обратите внимание, что Marshmallow представил [новую модель безопасности](https://developer.android.com/training/permissions/requesting.html) , в которой пользователи предоставляют разрешения во *время выполнения* , а не во время установки, и могут отзывать разрешения, когда захотят. Чтобы справиться с этим, вам нужно добавить код для запроса «опасных» разрешений (например, доступ к местоположению, телефону или SMS) каждый раз, когда вы его используете. Этот процесс обсуждается в лекции [«Файлы и разрешения»](https://info448.github.io/files-and-permissions.html#files-and-permissions) . Использование Интернета *не* является опасным разрешением, поэтому требуется только объявление разрешения в манифесте.

После того, как мы запросили разрешение (и получили его в силу того, что пользователь установил наше приложение), мы наконец можем подключиться к Интернету для загрузки данных. Мы можем вывести результаты запроса, чтобы доказать, что мы их получили!

Конечно, мы хотели бы отобразить эти данные на экране (а не просто вывести их). То есть, мы хотим поместить их в ListView, что означает, что нам нужно передать их обратно в Adapter (который работает для заполнения Views).

* Сначала очистите все предыдущие элементы данных в адаптере с помощью adapter.clear().
* Затем используйте adapter.add()или ( adapter.addAll()) для добавления каждого из новых элементов данных в модель адаптера! Обратите внимание, что вам может потребоваться выполнить синтаксический анализ данных в теле ответа, например, разбить строку или построить arrayили ArrayListиз данных JSON.
* Вы можете вызвать notifyDataSetChanged()адаптер, чтобы убедиться, что представление знает об изменении данных, но этот метод уже вызывается методом, .add()поэтому в данной ситуации он не нужен.

Вы можете использовать JsonObjectRequestкласс для загрузки данных как JSON-объекта, а не как необработанной строки. JSON-объекты и массивы можно преобразовать в Java-объекты/массивы с помощью двух классов: [JSONObject](http://developer.android.com/reference/org/json/JSONObject.html)и [JSONArray](http://developer.android.com/reference/org/json/JSONArray.html). Конструкторы для каждого из этих классов принимают JSON-строку, и вы можете вызвать getJSONArray(key)и , getJSONObject(key)чтобы получить вложенные объекты и массивы изнутри JSONObjectили JSONArray.

**ЗапросыОчередиОдиночные элементы**

Если вы собираетесь делать несколько сетевых запросов для своего приложения (что вы обычно и делаете для чего-либо разумного размера), то многократное создание новых RequestQueueобъектов будет расточительством — они могут занимать значительный объем памяти и мешать друг другу.

Вместо этого лучше всего использовать [шаблон проектирования Singleton](https://developer.android.com/training/volley/requestqueue.html#singleton) , чтобы гарантировать, что все ваше приложение использует только *один* RequestQueue.

Для этого вам нужно будет создать целый класс (например, VolleyService), который будет инстанцирован только один раз (это будет «синглтон»). Поскольку Volley RequestQueueконтролируется этим синглтоном, это означает, что будет только один RequestQueue.

*//java*

**public** **class** RequestSingleton { *//make static if an inner class!*

*//the single instance of this singleton*

**private** static RequestSingleton instance;

**private** RequestQueue requestQueue = **null**; *//the singleton's RequestQueue*

*//private constructor; cannot instantiate directly*

**private** RequestSingleton(Context ctx){

*//create the requestQueue*

**this**.requestQueue = Volley.newRequestQueue(ctx.getApplicationContext());

}

*//call this "factory" method to access the Singleton*

**public** static RequestSingleton getInstance(Context ctx) {

*//only create the singleton if it doesn't exist yet*

**if**(instance == **null**){

instance = **new** RequestSingleton(ctx);

}

**return** instance; *//return the singleton object*

}

*//get queue from singleton for direct action*

**public** RequestQueue getRequestQueue() {

**return** **this**.requestQueue;

}

*//convenience wrapper method*

**public** <T> void add(Request<T> req) {

requestQueue.add(req);

}

}

*//kotlin*

**private** **class** VolleyService

**private** **constructor**(ctx: Context) { *//private constructor; cannot instantiate directly*

**companion** **object** { *//to hold the shared instances*

**private** **var** instance: VolleyService? = **null** *//the single instance of this singleton*

*//call this "factory" method to access the Singleton*

**fun** getInstance(ctx: Context): VolleyService {

*//only create the singleton if it doesn't exist yet*

**if** (instance == **null**) {

instance = VolleyService(ctx)

}

**return** instance **as** VolleyService *//force casting*

}

}

*//from Kotlin docs*

**val** requestQueue: RequestQueue **by** lazy { *//instantiate once needed*

Volley.newRequestQueue(ctx.applicationContext) *//return the context-based requestQueue*

}

*//convenience wrapper method*

**fun** <T> add(req: Request<T>) {

requestQueue.add(req)

}

}

Для дополнительных примеров см. также демонстрацию Volley [на Kotlin от JetBrains.](https://github.com/JetBrains/kotlin-examples/tree/master/gradle/android-volley)

Такая структура позволит вам выполнять несколько сетевых запросов из нескольких компонентов вашего приложения, не пытаясь при этом задействовать несколько «диспетчеров», занимающих память.

**Загрузка изображений**

Помимо загрузки текстовых или JSON-данных через HTTP-запросы, Volley также поддерживает загрузку *изображений* для отображения в вашем приложении.

В целом, обработка изображений в Android — сложная задача. Изображения — это большие файлы (часто размером в несколько мегабайт), для загрузки которых может потребоваться обширная и длительная передача данных, а для отображения — интенсивное декодирование процессора. Поскольку мобильные устройства ограничены в *ресурсах* (особенно в памяти), попытка загрузить и отобразить множество изображений, например, в прокручиваемом списке, может быстро вызвать проблемы. См. [Обработка растровых изображений](https://developer.android.com/topic/performance/graphics/index.html) и [Эффективная загрузка больших растровых изображений](https://developer.android.com/topic/performance/graphics/load-bitmap.html) для некоторых примеров сложности, необходимой для работы с изображениями.

Volley предоставляет некоторую поддержку для облегчения загрузки и обработки изображений. В частности, он предоставляет встроенную поддержку для *управления сетью* (чтобы передача данных была максимально оптимизирована), *кэширования* (чтобы вам не приходилось дважды загружать одно и то же изображение) и для простого *отображения* загруженных по сети изображений.

Другие популярные библиотеки управления изображениями — [Glide ,](https://github.com/bumptech/glide)[Picasso](http://square.github.io/picasso/) от Square и [Fresco](https://github.com/facebook/fresco) от Facebook . Google рекомендует использовать Glide для выполнения сложных работ с изображениями. Однако, если загрузка изображений Volley достаточна для вашей задачи, это позволяет вам работать только с одной библиотекой и сетевой очередью.

Для эффективной загрузки изображений с помощью Volley вам необходимо настроить [ImageLoader](https://afzaln.com/volley/com/android/volley/toolbox/ImageLoader.html). Этот объект будет обрабатывать загрузку удаленных изображений, а также *кэшировать* их для будущего использования.

Чтобы создать экземпляр ImageLoader, вам необходимо предоставить , RequestQueueа также ImageCacheобъект, который представляет, как данные изображения должны кэшироваться (например, в памяти, на диске и т. д.). В документации Volley предлагается использовать объект для кэширования в памяти (хотя вы также LruCacheможете использовать ):DiskBasedCache

*//kotlin*

*//instantiate the image loader (from Kotlin docs)*

*//params are the requestQueue and the Cache*

**val** imageLoader: ImageLoader **by** lazy { *//only instantiate when needed*

ImageLoader(requestQueue,

**object** : ImageLoader.ImageCache { *//anonymous cache object*

**private** **val** cache = LruCache<**String**, Bitmap>(20)

**override** **fun** getBitmap(url: String): Bitmap? {

**return** cache.**get**(url)

}

**override** **fun** putBitmap(url: String, bitmap: Bitmap) {

cache.put(url, bitmap)

}

})

}

* Хорошей идеей будет сделать это ImageLoaderпеременной экземпляра VolleyService.

Получив ImageLoader, вы можете использовать его для загрузки изображения, вызвав его [get()](https://afzaln.com/volley/com/android/volley/toolbox/ImageLoader.html#get(java.lang.String,%20com.android.volley.toolbox.ImageLoader.ImageListener))метод и указав прослушиватель обратного вызова, который будет выполнен после завершения загрузки изображения.

Однако, вы почти всегда хотите загрузить изображение, чтобы отобразить его. Volley упрощает это, предоставляя настраиваемое Представление, называемое [NetworkImageView](https://afzaln.com/volley/com/android/volley/toolbox/NetworkImageView.html). A NetworkImageViewможет самостоятельно обрабатывать загрузку своего исходного изображения, интегрируя этот процесс в жизненный цикл Activity (например, поэтому оно не будет загружаться, когда Представление не отображается).

Вы объявляете a NetworkImageViewв XML-макете таким же образом, как указываете a ImageView:

**<com.android.volley.toolbox.NetworkImageView**

android:id="@+id/img\_remote"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:scaleType="fitXY"

**/>**

* Атрибут [android:scaleType](https://developer.android.com/reference/android/widget/ImageView.ScaleType.html)указывает, как следует масштабировать изображение, чтобы оно соответствовало представлению.

Чтобы загрузить изображение в это представление, вы вызываете setImageUrl()метод представления из вашего Java, указываете URL-адрес изображения для загрузки в представление и ImageLoaderдля использования для этого сетевого доступа:

**val** imageView = findViewById<NetworkImageView>(R.id.img\_remote)

netView.setImageUrl("https://ischool.uw.edu/fb-300x300.png", imageLoader);

И это позволит вам загружать и показывать изображения из Интернета (без необходимости превращать их в drawableресурс)!

1. <https://developer.android.com/guide/topics/ui/layout/listview.html>[↩](https://info448.github.io/data-views.html#fnref12)
2. <https://developer.android.com/reference/android/widget/Adapter.html>[↩](https://info448.github.io/data-views.html#fnref13)