

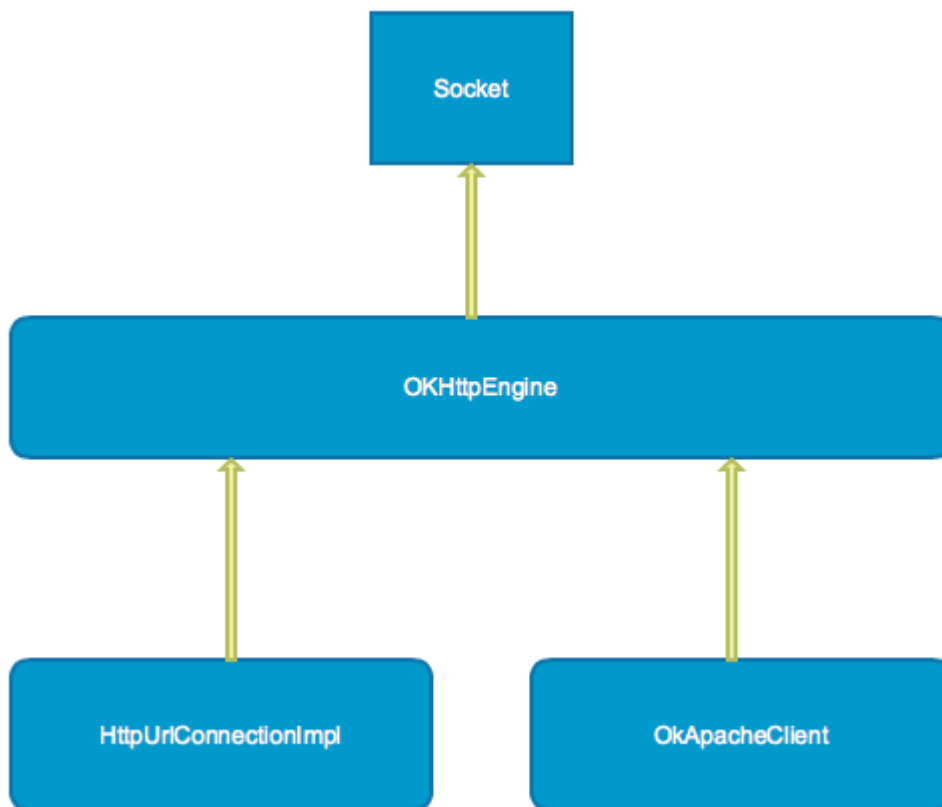
16. Работа с сетью. Дополнительные библиотеки

16.1 Работа с сетью

Сегодня почти все приложения используют HTTP/HTTPS запросы как своеобразный транспорт для своих данных. Даже если вы напрямую не используете эти протоколы, множество SDK (например, метрика, статистика падений, реклама), используют *HTTP/HTTPS* для работы с сетью.

В Android было два *HTTP* клиента: **HttpURLConnection** и **Apache HTTP Client** (`HttpURLConnection`). Команда Android в принципе не хотела активно работать над *Apache HTTP Client*.

В 2013 году *Square* выпустила **OkHttp**. **OkHttp** был создан для прямой работы с верхним уровнем сокетов Java, при этом не используя какие-либо дополнительные зависимости. Она поставляется в виде JAR-файла. Для упрощения перехода, *Square* реализовала *OkHttp* используя интерфейсы *HttpURLConnection* и *Apache client*.



OkHttp получила большое распространение и поддержку сообществом, и, Google решили использовать версию 1.5 в *Android 4.4 (KitKat)*. В 2015 Google официально признала *AndroidHttpClient*, основанный на *Apache*, устаревшим, вместе с выходом *Android 5.1 (Lollipop)*.

Сегодня **OkHttp** поставляется со следующим набором функций:

1. Поддержка HTTP/2 и SPDY позволяет всем запросам, идущим к одному хосту, делиться сокетом
2. Объединение запросов уменьшает время ожидания (если SPDY не доступен)
3. Прозрачный GZIP позволяет уменьшить вес загружаемой информации
4. Кэширование ответов позволяет избежать работу с сетью при повторных запросах.
5. Поддержка как и синхронизированных блокирующих вызовов, так и асинхронных вызовов с обратным вызовом (callback)

```
private val client = OkHttpClient()

@Throws(Exception::class)
fun run() {
    val request = Request.Builder()
        .url("http://publicobject.com/helloworld.txt")
        .build()

    client.newCall(request).enqueue(object : Callback() {
        fun onFailure(request: Request, throwable: Throwable) {
            throwable.printStackTrace()
        }

        @Throws(IOException::class)
        fun onResponse(response: Response) {
            if (!response.isSuccessful())
                throw IOException("Unexpected code $response")
            System.out.println(response.body().string())
        }
    })
}
```

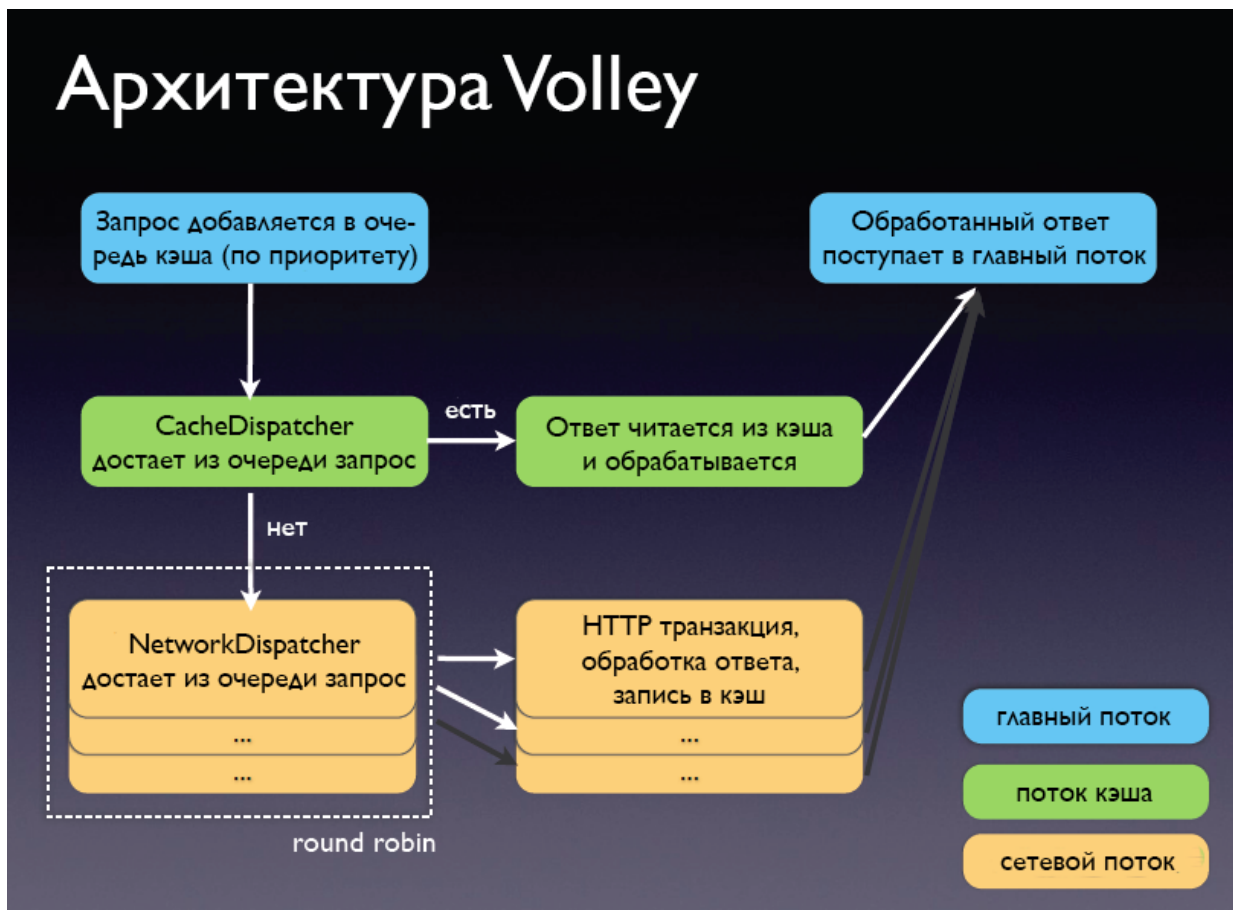
Работа с сетью не должна быть в UI потоке. Начиная с *Android 3.0 (Honeycomb, API 11)*, работа с сетью в отдельном потоке стала обязательной. Для *HttpURLConnection* потребовалось бы построить конструкцию с использованием *AsyncTask* или отдельного потока. Сложности возникнут если добавить отмену загрузки, объединение соединений и т.п.

16.2 Библиотеки для работы с сетью

Volley

HTTP библиотека от Google. Преимущества:

1. Автоматическое планирование сетевых запросов
2. Множество параллельных сетевых соединений
3. Прозрачное кэширование в памяти и на диске, в соответствии со стандартной кэш-согласованностью.
4. Поддержка приоритизации запросов.
5. Отмена API запросов. Вы можете отменить как один запрос, так и целый блок.
6. Простота настройки, например, для повторов и отсрочек.
7. Строгая очередность, которая делает легким корректное заполнение данными, полученными асинхронно из сети, интерфейса пользователя.
8. Инструменты отладки и трассировки



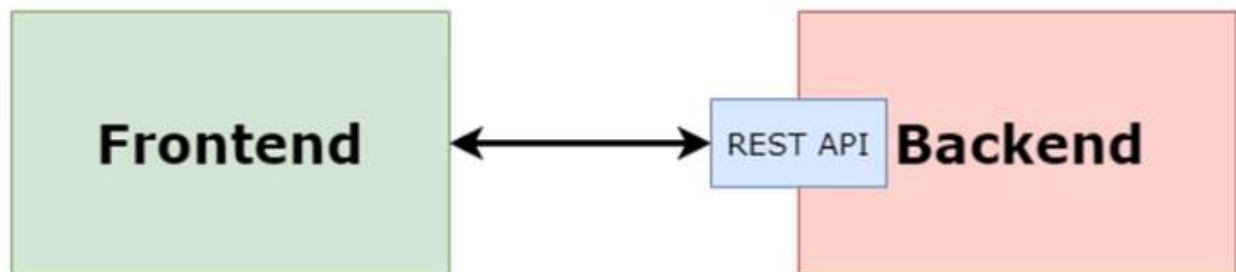
Если вы хотите получить *JSON* или изображение, то *Volley* имеет на это специальный абстракции, такие как *ImageRequest* и *JsonObjectRequest*, которые помогают вам в автоматическом режиме конвертировать полезную нагрузку *HTTP*.

HTTP клиенты продолжили развиваться для поддержки приложений с большим количеством картинок. В то же время, REST API стал стандартом в индустрии, и каждый разработчик имел дело с такими типовыми задачами как сериализация в/из JSON и преобразование REST-вызовов в интерфейсы Java. Поэтому появились библиотеки, решающие эти задачи:

- **Retrofit** – типобезопасный HTTP Android клиент для взаимодействия с REST-интерфейсами
- **Picasso** – библиотека для загрузки и кэширования изображений под Android

Понятие REST

REST (сокращение от Representational State Transfer — «передача состояния представления») — это архитектурный стиль взаимодействия приложений. REST представляет собой согласованный набор ограничений, учитываемых при проектировании распределённой гипермедиа-системы. В определённых случаях (интернет-магазины, поисковые системы, прочие системы, основанные на данных) это приводит к повышению производительности и упрощению архитектуры. У *REST* отсутствует какой-либо стандарт, а данные между клиентом и сервером могут передаваться в любом виде, будь то JSON, XML, YAML и т.д.



REST просит разработчиков использовать методы *HTTP* явно и таким образом, чтобы это соответствовало определению протокола. Этот основной принцип проектирования REST устанавливает взаимно-однозначное сопоставление между операциями создания, чтения, обновления и удаления (CRUD) и методами HTTP. Согласно этому отображению:

- Чтобы создать ресурс на сервере, используйте POST.
- Чтобы получить ресурс, используйте GET.
- Чтобы изменить состояние ресурса или обновить его, используйте PUT.
- Чтобы удалить или удалить ресурс, используйте DELETE.



Picasso

<https://square.github.io/picasso/>

Предоставляет HTTP библиотеку, ориентированную на работу с изображениями. Например, вы можете загрузить изображение в свой View с помощью одной строчки:

```
Picasso.with(context).load("http://i.imgur.com/DvvpvklR.png").into(imageView);
```

Glide — похоже на Picasso. Он предоставляет некоторые дополнительные функции
<https://www.glideapps.com/>

Fresco

<https://github.com/facebook/fresco>

Используется в мобильном приложении Facebook. Одна из ключевых функций, которая выделяет ее, — кастомная стратегия выделения памяти для bitmap'ов, чтобы избежать работы долгого GC (сборщик мусора). Fresco выделяет память в регионе, который называется *ashmem*. Используются некие трюки, чтобы иметь доступ к этому региону памяти доступ как из части, написанной на C++, так и из части на Java. Чтобы уменьшить потребление CPU и данных из сети, эта библиотека использует 3 уровня кэша: 2 в ОЗУ, третий во внутреннем хранилище.

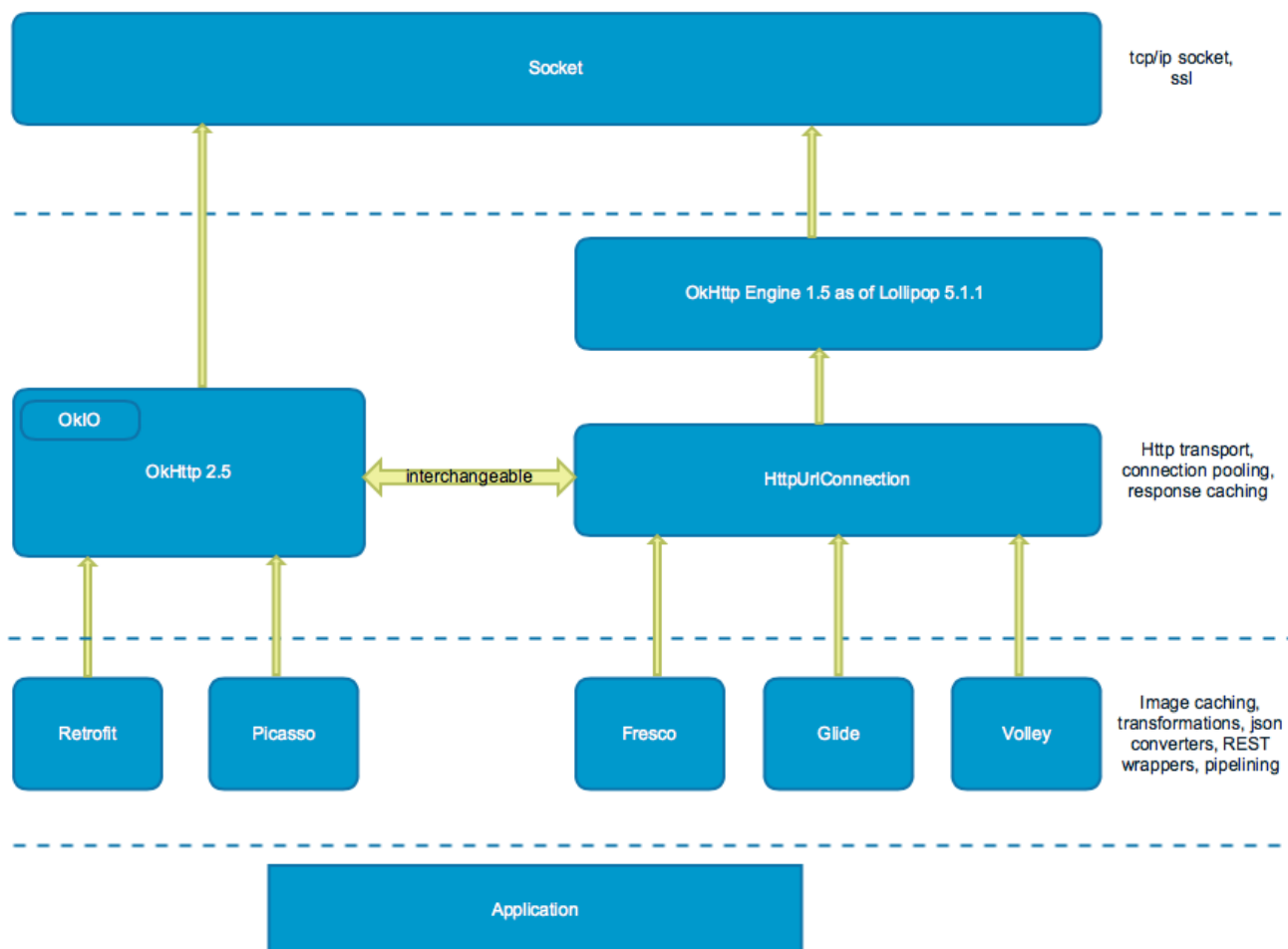
Retrofit

<https://square.github.io/retrofit/>

Retrofit — это REST клиент для Java и Android. Он позволяет легко получить и загрузить JSON (или другие структурированные данные) через веб-сервис на основе REST. В *Retrofit* вы настраиваете, какой конвертер используется для сериализации данных. Обычно для JSON используется GSON, но вы можете добавлять собственные конвертеры для обработки XML или других протоколов. В *Retrofit* используется библиотека *OkHttp* для HTTP-запросов.

Каждый метод интерфейса представляет собой один из возможных вызовов API. Он должен иметь HTTP аннотацию (GET, POST и т. д.), чтобы указать тип запроса и относительный URL. Возвращаемое значение завершает ответ в Call-объекте с типом ожидаемого результата.

Рассмотренные библиотеки связаны следующим образом



Как вы можете увидеть, HTTP всегда остается внизу у высокоуровневых библиотек. Вы можете выбирать между простым *URLConnection* или последним *OkHttpClient*.

16.3 Работа с библиотекой OkHttp 3 (пример)

Чтобы выполнять сетевые вызовы, можно использовать библиотеку **OkHttp**. Нужно добавить зависимость **OkHttp 3.10** в файл `build.gradle`:

```
implementation 'com.squareup.okhttp3:okhttp:3.10.0'
```

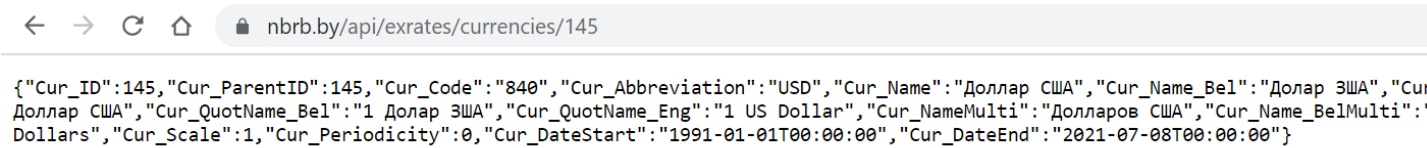
Также для выполнения сетевых вызовов, нужно добавить разрешение *INTERNET* в манифест приложения:

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

Для интерфейса добавим *ImageView*, *TextView*, в котором будем отображать данные, кнопку, которая позволит загружать данные запрашивая его из базы данных, *ProgressBar*.

Перед написанием кода в *MainActivity* мы протестируем ответ, возвращаемый API. Будем обращаться по следующему адресу: <https://www.nbrb.by/api/exrates/currencies>

Этот веб-сервис возвращает следующий результат:



← → ↺ 🏠 nbrb.by/api/exrates/currencies/145

```
{
  "Cur_ID":145,"Cur_ParentID":145,"Cur_Code":"840","Cur_Abbreviation":"USD","Cur_Name":"Доллар США","Cur_Name_Bel":"Долар ЗША","Cur_Name_Rus":"Доллар США","Cur_QuotName_Bel":"1 Долар ЗША","Cur_QuotName_Eng":"1 US Dollar","Cur_NameMulti":"Долларов США","Cur_Name_BelMulti":"Долары","Cur_Scale":1,"Cur_Periodicity":0,"Cur_DateStart":"1991-01-01T00:00:00","Cur_DateEnd":"2021-07-08T00:00:00"}

```

<https://www.nbrb.by/api/exrates/rates/431>

В *MainActivity* определяем переменную, в которой храним *URL* конечной точки API, который мы собираемся вызвать. Затем создаем экземпляр объекта *OkHttpClient*.

В методе *onCreate MainActivity* просто нужно установить *OnClickListener* на кнопку.

Обращение к API выполняется в специальном методе *loadRandomCurr*. Отображаем *ProgressBar* непосредственно перед обращением к сети. Затем создаём объект *Request* с *URL*-адресом конечной точки в параметре.

После этого мы вызываем метод *newCall* на *OkHttpClient*, передавая в него *Request* в качестве параметра. Чтобы обработать ответ, вызываем метод *enqueue* с экземпляром *Callback* в параметре.

В методе *onResponse* мы получаем ответ и затем создаём *JSONObject*. Последний шаг — получить свойство объекта. После этого мы можем отобразить в *TextView*, инкапсулировав всё в блок *runOnUiThread*, чтобы быть уверенным, что обновление пользовательского интерфейса будет выполнено в потоке пользовательского интерфейса.

```
import android.view.View
import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.text.Html
import kotlinx.android.synthetic.main.activity_main.*
import okhttp3.*
import org.json.JSONObject
import java.io.IOException

class MainActivity : AppCompatActivity() {

    val URL = "https://www.nbrb.by/api/exrates/rates/431"
    var okHttpClient: OkHttpClient = OkHttpClient()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        nextBtn.setOnClickListener {
            loadRandomCurr()
        }
    }
}
```

```

private fun loadRandomCurr() {
    runOnUiThread {
        progressBar.visibility = View.VISIBLE
    }

    val request: Request = Request.Builder()
        .url(URL)
        .build()
    OkHttpClient.newCall(request).enqueue(object: Callback {
        override fun onFailure(call: Call?, e: IOException?) {

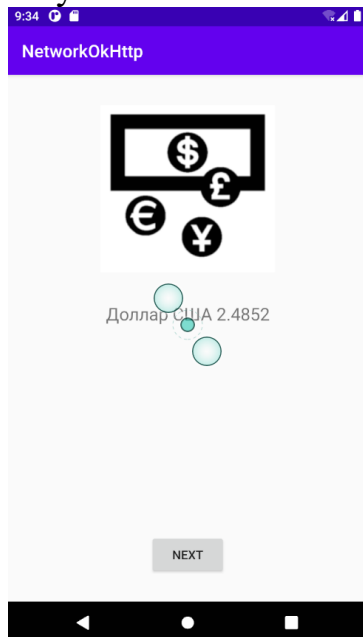
        }

        override fun onResponse(call: Call?, response: Response?) {
            val json = response?.body()?.string()
            val txt = (JSONObject(json)
                .get("Cur_Name")).toString()
            val txt2 = (JSONObject(json)
                .get("Cur_OfficialRate")).toString()

            runOnUiThread {
                progressBar.visibility = View.GONE
                curr.text = Html.fromHtml(txt+" " +txt2)
            }
        }
    })
}
}

```

Звпускаем



16.4 Использование Retrofit 2.x в качестве REST клиента

<https://square.github.io/retrofit/>

Добавление зависимостей

Добавьте следующую зависимость в файл build.gradle:

```
implementation 'com.squareup.retrofit2:retrofit:2.4.0'
```

Retrofit предоставляет зависимость, которая автоматически конвертирует JSON в POJO. Для этого добавьте ещё одну зависимость в файл build.gradle:

```
implementation 'com.squareup.retrofit2:converter-gson:2.3.0'
```

Затем вам может понадобится:

Okhttp — для создания HTTP-запроса с соответствующими заголовками.

Retrofit — для создания запроса

Moshi / GSON — для парсинга данных JSON

Kotlin Coroutines — для создания сетевых запросов, не блокирующих основной поток.

Picasso / Glide — для скачивания и установки изображения в ImageView.

Поэтому добавляем (версии возможно будут другие):

```
implementation "com.jakewharton.retrofit:retrofit2-kotlin-coroutines-adapter:0.9.2"
implementation "com.squareup.okhttp3:okhttp:3.12.0"
implementation 'com.squareup.okhttp3:logging-interceptor:3.11.0'
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-android:1.0.1"
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:1.0.1"
implementation 'com.squareup.retrofit2:converter-moshi:2.4.0'
```

Добавьте строку разрешения работы с сетью в файл AndroidManifest:

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

Общий принцип работы

Retrofit - это класс, с помощью которого ваши интерфейсы API превращаются в вызываемые объекты. По умолчанию *Retrofit* даст вам приемлимые значения по умолчанию для вашей платформы, но можно и настраивать.

По умолчанию Retrofit может десериализовать только HTTP-тела в тип *ResponseBody* *OkHttp*, и он может принимать только тип *RequestBody* для **@Body**.

Конвертеры могут быть добавлены для поддержки других типов. Шесть модулей адаптируют популярные библиотеки сериализации.

- Gson: com.squareup.retrofit2:converter-gson
- Jackson: com.squareup.retrofit2:converter-jackson
- Moshi: com.squareup.retrofit2:converter-moshi
- Protobuf: com.squareup.retrofit2:converter-protobuf
- Wire: com.squareup.retrofit2:converter-wire
- Simple XML: com.squareup.retrofit2:converter-simplexml

Android networking работает следующим образом:

Request — Выполнение HTTP-запроса к URL-адресу (конечной точке — endpoint) с соответствующими заголовками. При необходимости используется ключ авторизации.

Response — Запрос возвращает ответ, содержащий error или success. В случае успешного завершения запроса, ответ содержит содержимое endpoint (обычно в формате JSON)

Parse & Store — Парсинг JSON, получение необходимых значений и размещение их в классе данных.

Для работы с Retrofit вам понадобится следующее:

- Model класс, который используется как модель JSON
- Интерфейсы, которые определяют возможные HTTP операции
- Класс Retrofit.Builder — экземпляр, который использует интерфейс и API Builder, чтобы задать определение конечной точки URL для операций HTTP

Обычно сервер предоставляет API, т.е. набор методов (он же REST).

Пример

Возьмем конечные точки, которые возвращают сообщение. Конечная конечная точка принимает GET-запрос (или другой тип) и возвращает данные в формате JSON (или другом)

<https://jsonplaceholder.typicode.com>

JSONPlaceholder - это бесплатный онлайн-REST API, который можно использовать всякий раз, когда вам нужны fake данные. Например, просто для локального тестирования.

JSONPlaceholder содержит 6 ресурсов:

/posts	-	100 posts
/comments		500 comments
/albums		100 albums
/photos		5000 photos
/todos		200 todos
/users		10 users

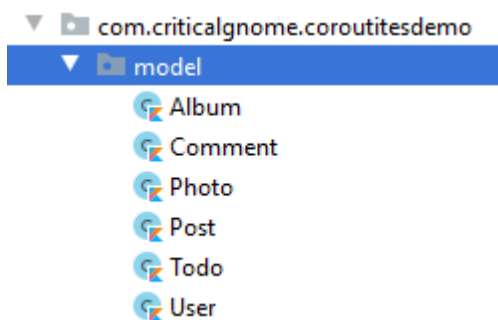
ресурсы имеют отношения. Например: сообщения имеют много комментариев, альбомы содержат много фотографий, ... полный список см. В руководстве.

Поддерживаются все HTTP методы а также http и https дял запросв.

```
GET    /posts
GET    /posts/1
GET    /posts/1/comments
GET    /comments?postId=1
POST   /posts
PUT    /posts/1
PATCH /posts/1
DELETE /posts/1
```

Для работы с **Retrofit** понадобятся три класса.

1. POJO (Plain Old Java Object) или Model Class - json-ответ от сервера нужно реализовать как модель
2. Retrofit - класс для обработки результатов. Ему нужно указать базовый адрес в методе **baseUrl()**
3. Interface - интерфейс для управления адресом, используя команды GET, POST и т.д.



Далее описываем интерфейс

```
interface JsonPlaceholderApi {

    @GET("/posts")
    fun getPosts(): Deferred<Response<List<Post>>>

    @GET("/comments")
    fun getComments(): Deferred<Response<List<Comment>>>

    @GET("/albums")
    и т.д.
```

В интерфейсе задаются команды-запросы для сервера. Команда комбинируется с базовым адресом сайта (baseUrl()) и получается полный путь к странице.

Создаем объект компаньен. Объект для запроса к серверу создаётся в простейшем случае следующим образом

```
companion object {
    private const val BASE_URL = "https://jsonplaceholder.typicode.com"
    fun getApi(): JsonPlaceholderApi {
        return Retrofit.Builder()
            .baseUrl(BASE_URL)
            .addConverterFactory(MoshiConverterFactory.create())
            .addCallAdapterFactory(CoroutineCallAdapterFactory())
            .client(getClient())
            .build()
            .create(JsonPlaceholderApi::class.java)
    }

    private fun getClient(): OkHttpClient {
        val client = OkHttpClient.Builder()
        if (BuildConfig.DEBUG) {
            val logging = HttpLoggingInterceptor()
            logging.level = HttpLoggingInterceptor.Level.BODY
            client.addInterceptor(logging)
        }
        return client.build()
    }
}
```

В результате библиотека Retrofit сделает запрос, получит ответ и производит разбор ответа, раскладывая по полочкам данные. Вам остаётся только вызывать нужные методы класса-модели для извлечения данных.

HttpLoggingInterceptor является частью OkHttpClient, но поставляется отдельно от неё. Перехватчик следует использовать в том случае, когда вам действительно нужно изучать логи ответов сервера. По сути библиотека является сетевым аналогом привычного LogCat.

Подключаем перехватчик к веб-клиенту. Добавляйте его после других перехватчиков, чтобы ловить все сообщения. Существует несколько уровней перехвата данных: NONE, BASIC, HEADERS, BODY. Последний вариант самый информативный, пользуйтесь им осторожно. При больших потоках данных информация забьёт весь экран. Используйте промежуточные варианты.

```
class MainAdapter(var items: List<Post>) : RecyclerView.Adapter<MainAdapter.PostHolder>() {
    override fun getItemCount() = items.size
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int) = PostHolder(LayoutInflater.from(parent.context).inflate(R.layout.post_item, parent, false))
    override fun onBindViewHolder(holder: PostHolder, position: Int) =
        holder.bind(items[position])

    inner class PostHolder(view: View) : RecyclerView.ViewHolder(view) {
        fun bind(item: Post) = with(itemView) {
```

```

        postId.text = item.id.toString()
        postTitle.text = item.title
        postBody.text = item.body
    }
}
}

```

Выводить будем в RecyclerView и делать привязку данных

```

class MainActivity : Activity() {

    private val jsonPlaceholderApi = JsonPlaceholderApi.getApi()
    private val adapter = MainAdapter(listOf())

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        recycler.adapter = adapter

        GlobalScope.launch(Dispatchers.Main) {
            progress.visibility = View.VISIBLE
            val postsResponse = jsonPlaceholderApi.getPosts().await()
            progress.visibility = View.GONE
            if (postsResponse.isSuccessful) {
                adapter.items = postsResponse.body() ?: listOf()
                adapter.notifyDataSetChanged()
            } else {
                Toast.makeText(this@MainActivity, "Error ${postsResponse.code()}",
                    Toast.LENGTH_SHORT).show()
            }
        }
    }
}

```

