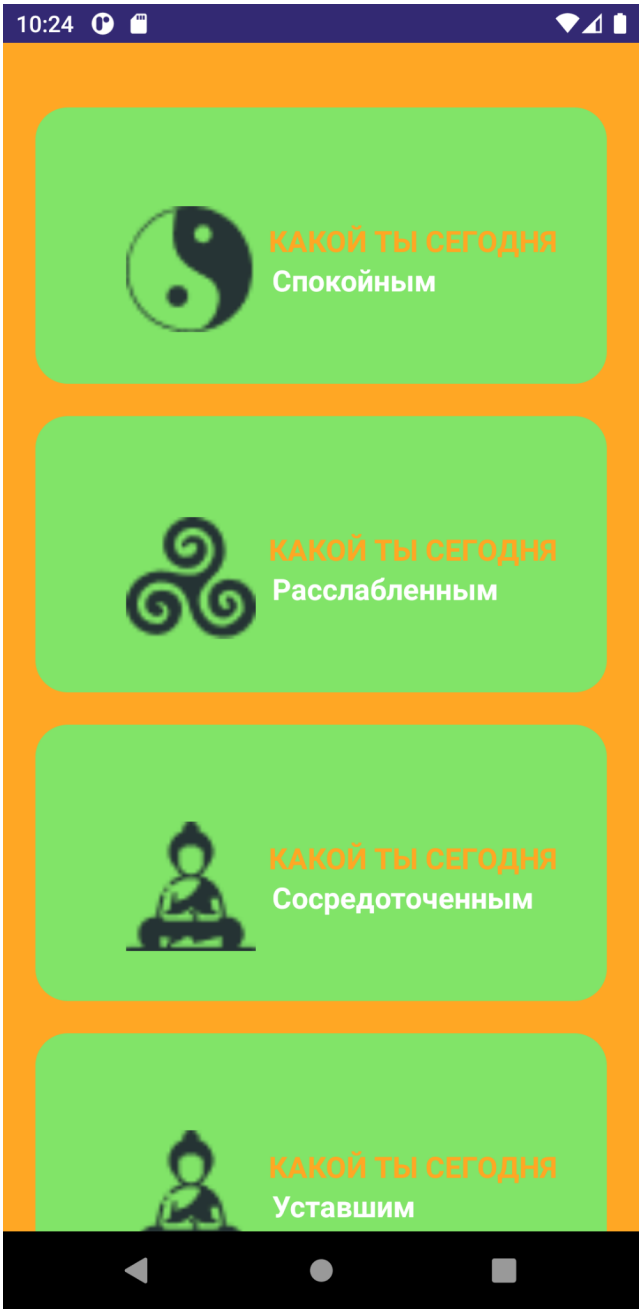
Практическая работа 12
“Работа с REST API. RETROFIT”



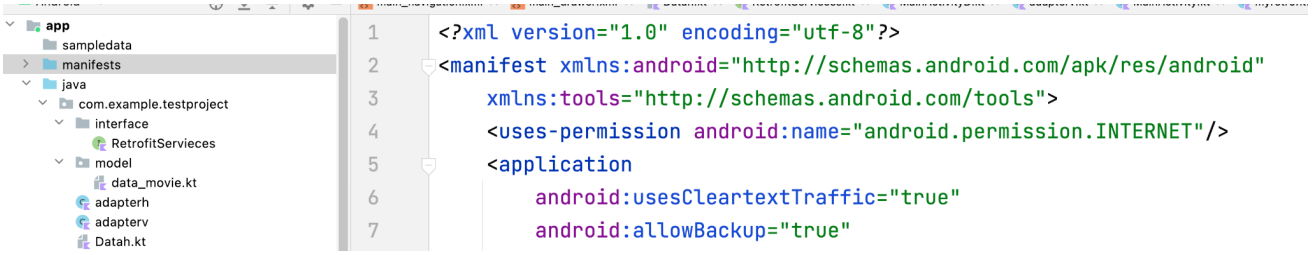
http://mskko2021.mad.hakta.pro/api - начальный URL к API

Продолжите работу с приложением Cinema.

Для корректной работы требуется разрешить приложению доступ к интернету. Добавьте указанную строку кода в файл манифеста.

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

и очистку траффика (строки 4 и 6)

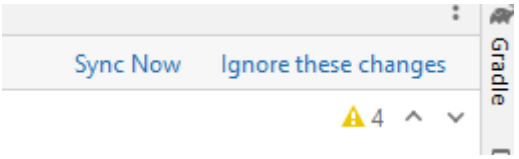


Для работы с библиотекой Retrofit необходимо подключить в проект следующие зависимости:

implementation 'com.squareup.retrofit2:retrofit:2.9.0' - подключение библиотеки

implementation 'com.squareup.retrofit2:converter-gson:2.5.0' - Конвертер JSON в POJO

Добавьте указанные зависимости в файл build.gradle и пересоберите проект, нажав Sync Now:



*Библиотека **GSON** была разработана программистами Google и позволяет конвертировать объекты JSON в Java-объекты и наоборот.*

POJO – это plain old Java object, простой Java-объект, не ограниченный какими-либо запретами, специфичными для того или иного фреймворка (за исключением спецификации самой Java, разумеется) и пригодный для использования в любой среде.

POJO используются для универсальной и наглядной сериализации и десериализации данных. Основное их назначение – это обработка Json в теле запросов и ответов.

Они позволяют достаточно гибко настраивать содержимое Json-объектов, отправляемых в теле запросов – например можно использовать один POJO-класс для составления Json-объектов с разным набором значений, избежав при этом многократного повторения одного и того же кода, или легко создавать Json-объекты с многоуровневыми вложениями, сохраняя при этом простую и наглядную структуру.

При обработке входящих ответов POJO позволяют извлекать любые необходимые значения для дальнейшего использования, а также дают (как уже было сказано выше – простой и наглядный) доступ к вложенным значениям.

https://qa.crtweb.ru/docs/testing/auto/api/pojo/

Парсинг (parsing) — это сбор информации из сторонних источников для использования полученных данных в различных целях, от аналитики до копирования. Парсинг - это процесс автоматического сбора данных и их структурирования. Парсинг обычно применяют, когда нужно быстро собрать большой объем данных. Его выполняют с помощью специальных сервисов — парсеров

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

1. POJO (Plain Old Java Object) или Model Class - json-ответ от сервера нужно реализовать как модель. **Model** — это логика, которая связана с данными приложения. Другими словами это POJO, классы работы с API, базой данных.
2. Interface - интерфейс для управления адресом, используя команды GET, POST и т.д.
3. Retrofit - класс для обработки результатов. Ему нужно указать базовый адрес в методе **baseUrl()**

По запросу к серверу мы будем получать изображения и цитаты.

Необходимо создать дата класс, с помощью которых будем получать данные с сервера (идентификатор, цитату и изображение). Для этого необходимо посмотреть на структуру ответа в документации API в виде JSON (или других форматов) и создать на его основе дата класс.

Создайте папку и назовите ее model, внутри папки создайте data class, назовите его data_movie. В круглых скобках укажите поля класса, они должны по

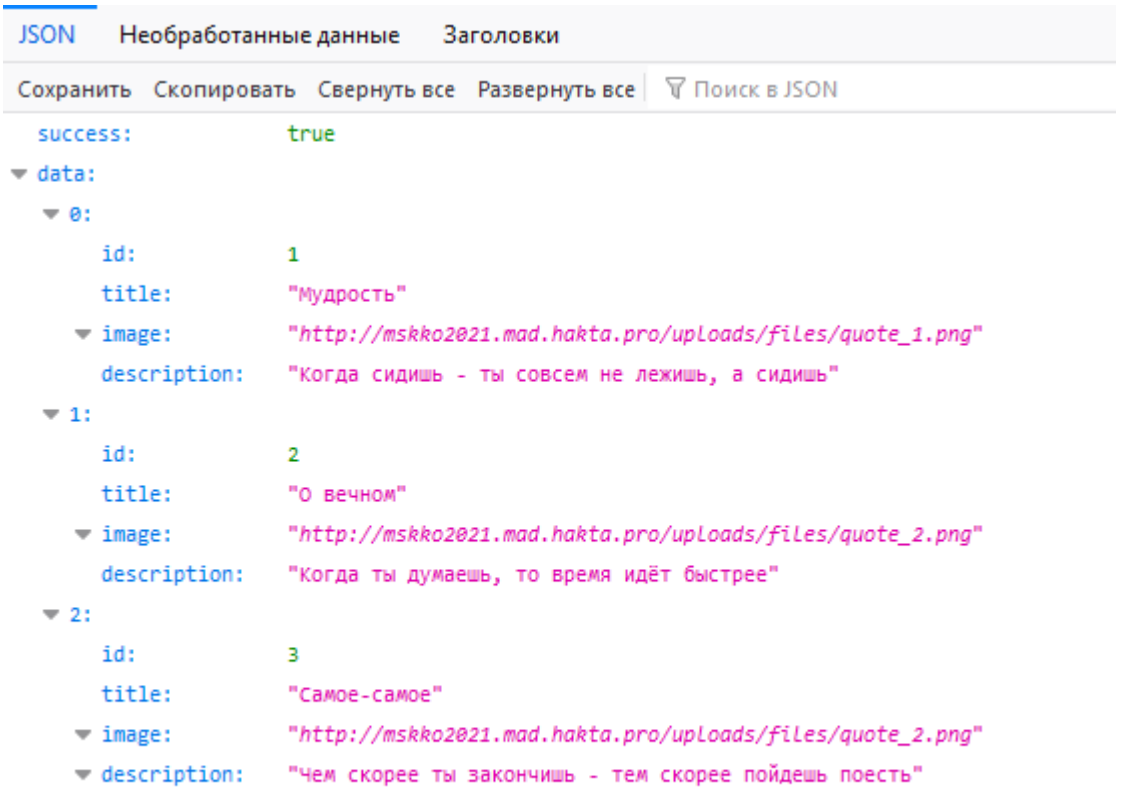
названию и типу соответствовать объектам JSON в спецификации:

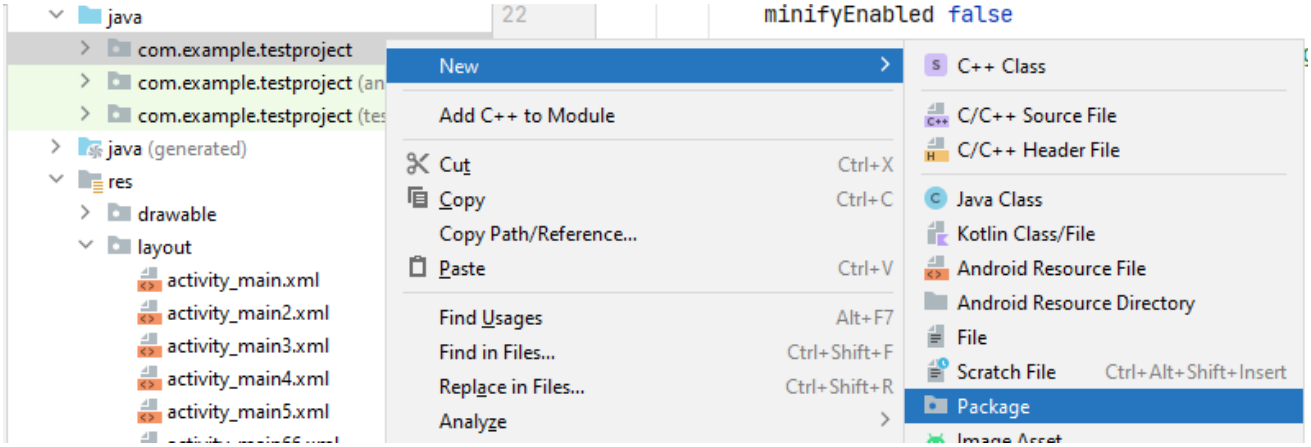
API Url: http://mskko2021.mad.hakta.pro/api

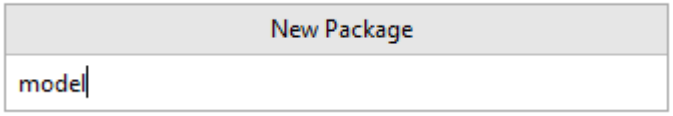
API Default Construction:

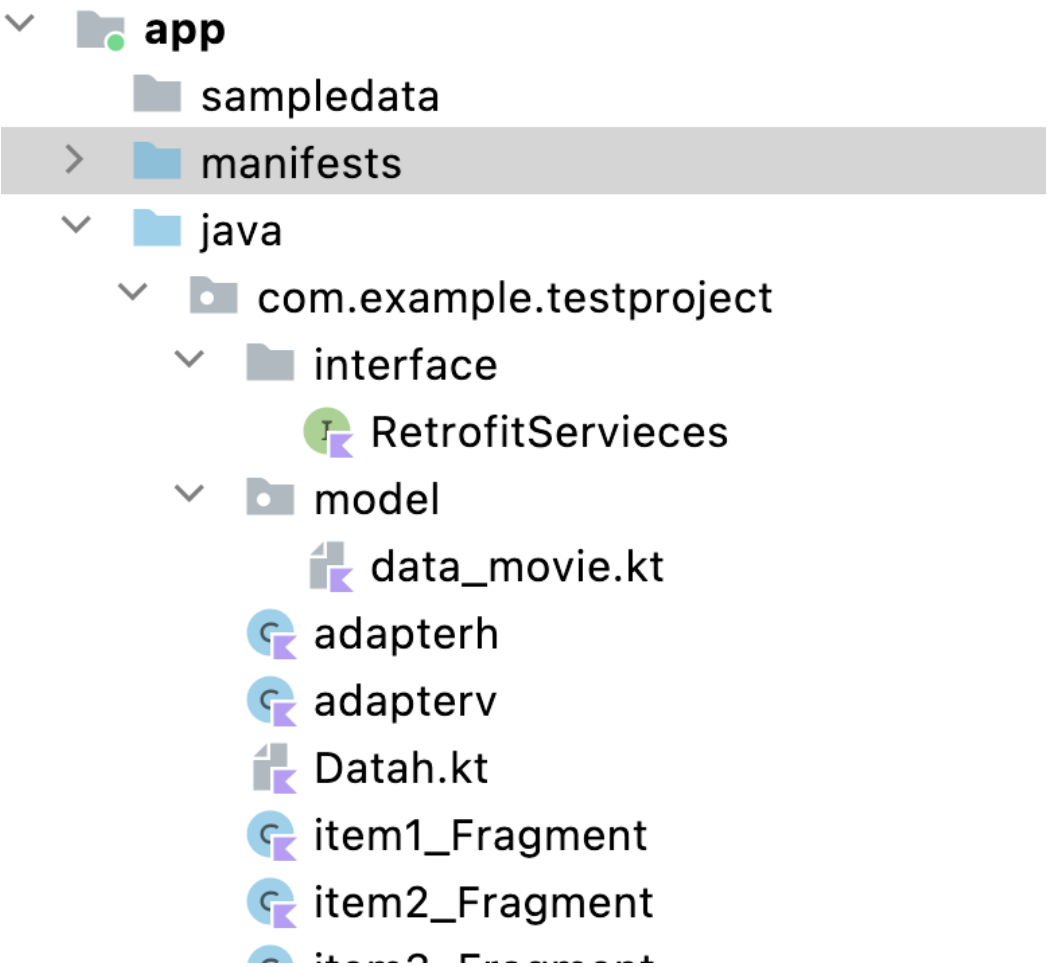
```
{
"success": boolean,
"data": JSONArray/JSONObject,
"debug" : JSONObject
}
```

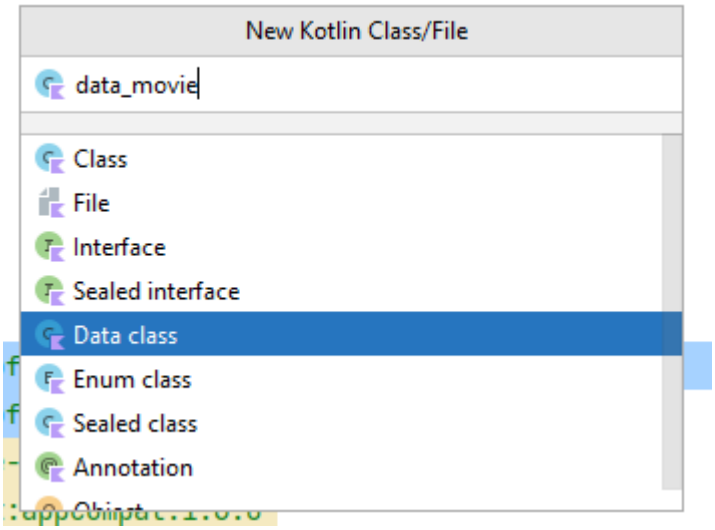
Method	URL	Params	Description	Result
GET	/ping			
GET	/quotes	string $endpoint	Returns array of quotes	
GET	/feelings	string $endpoint	Returns array of quotes	
POST	/user			

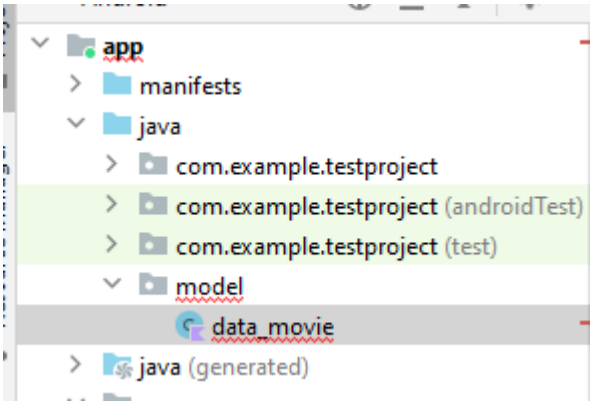



java
com.example.testproject
com.example.testproject (an
com.example.testproject (tes
java (generated)
res
drawable
layout
activity_main.xml
activity_main2.xml
activity_main3.xml
activity_main4.xml
activity_main5.xml
22
minifyEnabled false
New
Add C++ to Module
Cut Ctrl+X
Copy Ctrl+C
Copy Path/Reference...
Paste Ctrl+V
Find Usages Alt+F7
Find in Files... Ctrl+Shift+F
Replace in Files... Ctrl+Shift+R
Analyze
C++ Class
C/C++ Source File
C/C++ Header File
Java Class
Kotlin Class/File
Android Resource File
Android Resource Directory
File
Scratch File Ctrl+Alt+Shift+Insert
Package


New Package
model


app
sampledata
manifests
java
com.example.testproject
interface
RetrofitServieces
model
data_movie.kt
adapterh
adapterv
Datah.kt
item1_Fragment
item2_Fragment





API Url: http://mskko2021.mad.hakta.pro/api

API Default Construction:

```
{
"success": boolean,
"data": JSONArray/JSONObject,
"debug" : JSONObject
}
```

Method	URL	Params	Description	Result
GET	/ping			
GET	/quotes	string $endpoint	Returns array of quotes	
GET	/feelings	string $endpoint	Returns array of quotes	
POST	/user			

Создайте Api интерфейс и заполните его запросами.

В интерфейсе задаются команды-запросы для сервера. Команда

комбинируется с базовым адресом сайта/сервера (**baseUrl()**) и получается полный путь к странице. Код может быть простым и сложным. Запросы размещаются в абстрактном классе **Call** с указанием желаемого типа. **Interface -** нужен для создания абстрактных классов.

Интерфейсы схожи с абстрактными классами и предоставляют лишь методы без реализации. В интерфейсах можно записать методы, что должны реализовываться во всех классах, использующих интерфейс.

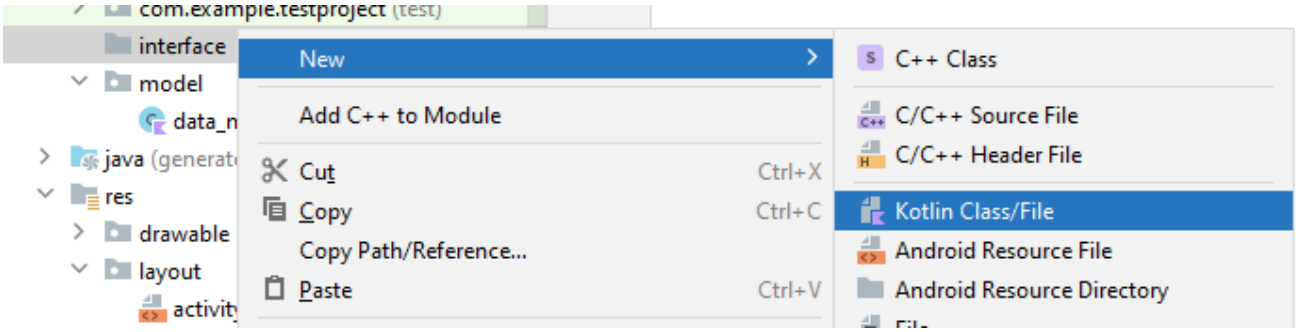
https://swiftbook.ru/post/tutorials/interfaces-and-abstract-classes-in-kotlin/ - про абстрактные классы

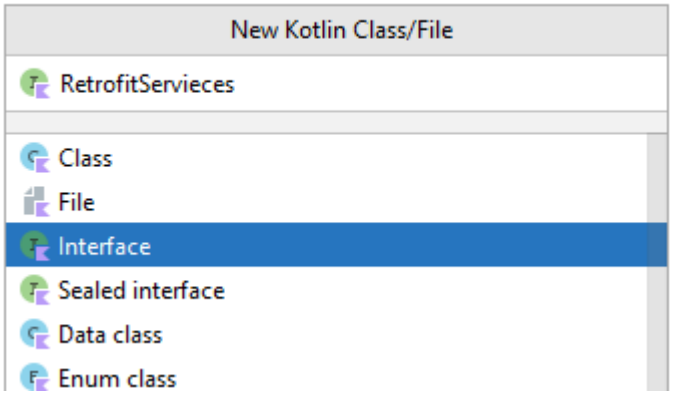
https://itproger.com/course/kotlin/8 - про интерфейсы

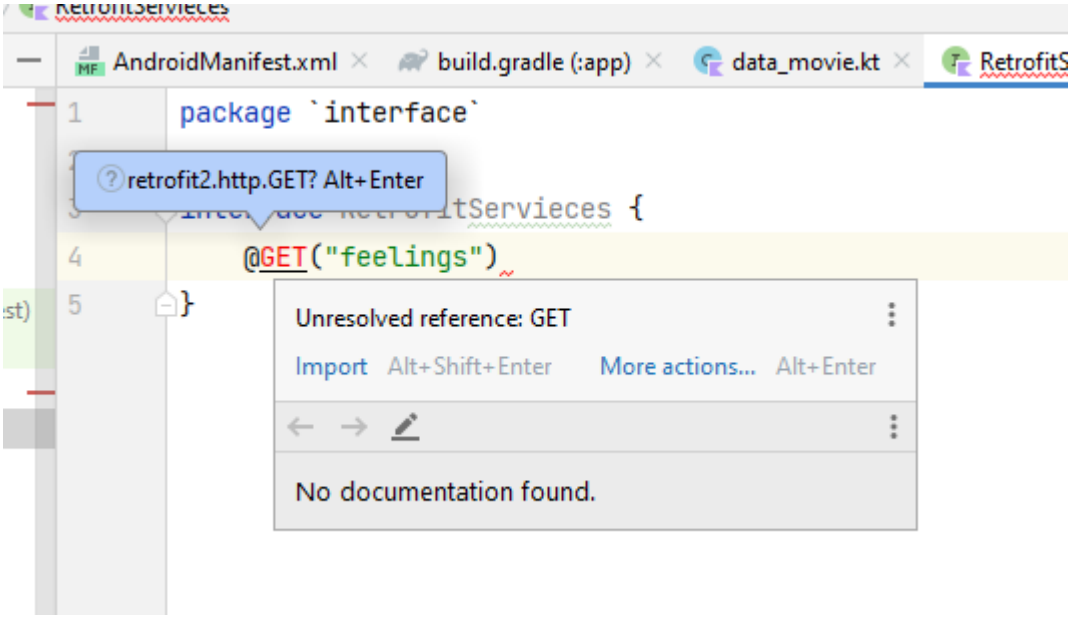
Создайте пакет interface в в нем создайте файл interface, назвав его RetrofitServieces. Пропишите @GET запрос в скобках напишите кавычки, а в кавычках укажите ветку, с которой будут парсить данные - feelings.

***GET** — запрашивает данные с определенного ресурса(сайта)*

***POST** — отправляет данные на сервер для последующей обработки*

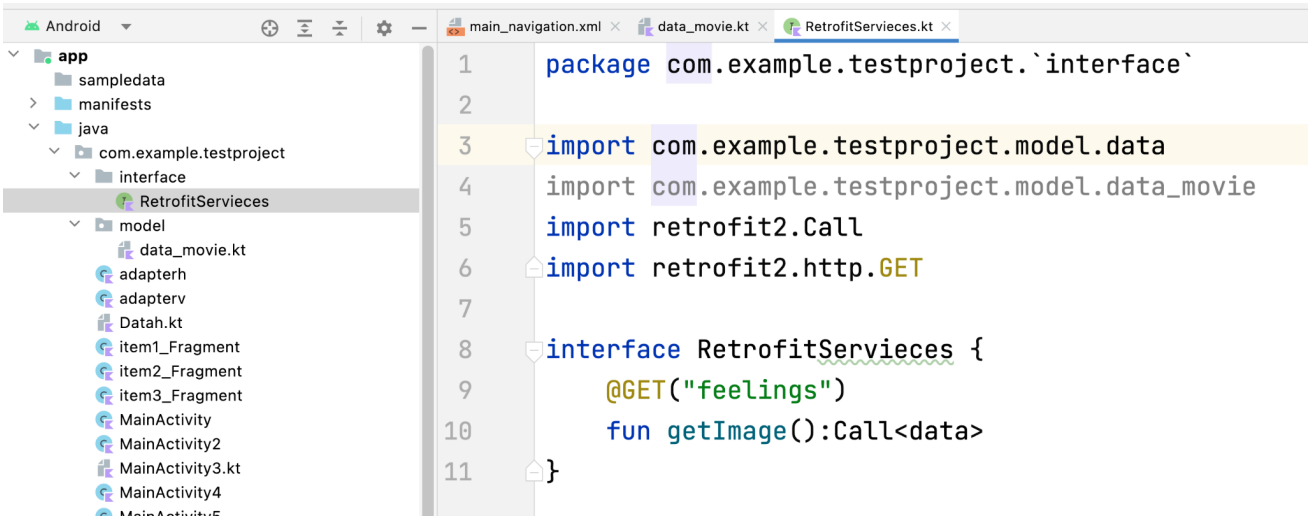




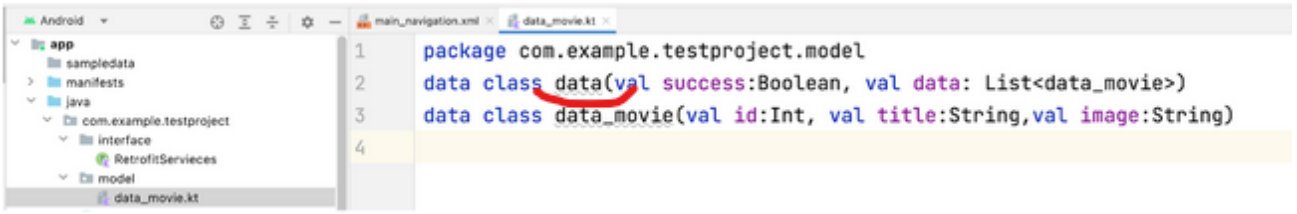


Пропишем функцию getImage, которая должна будет возвращать Call типа data_movie (имя созданного ранее дата класса).

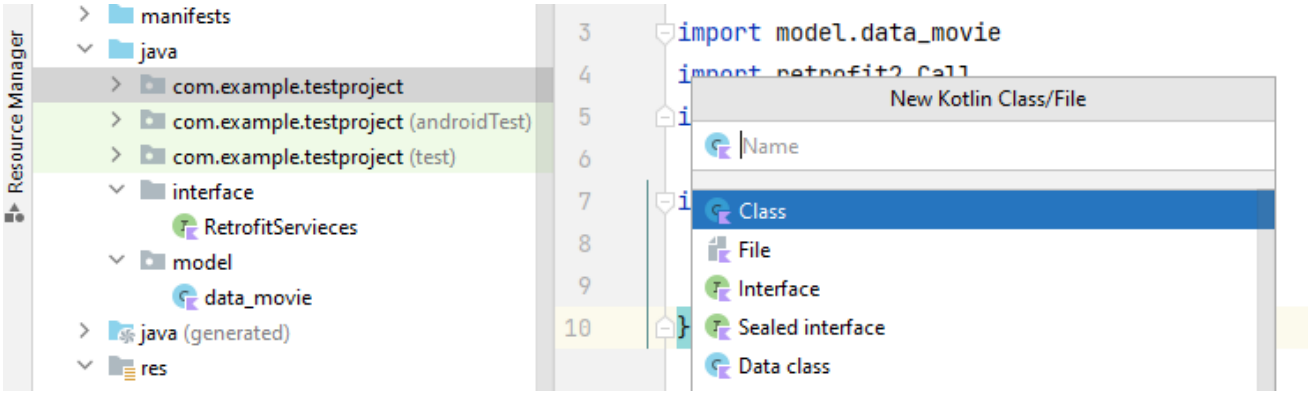
При создании функции обращения к REST API возвращаемым значением необходимо указать класс Call, который сформирует вызов. Обратите внимание при импорте библиотеки необходимо выбрать библиотеку Call (retrofit2) иначе у вас возникнут ошибки.

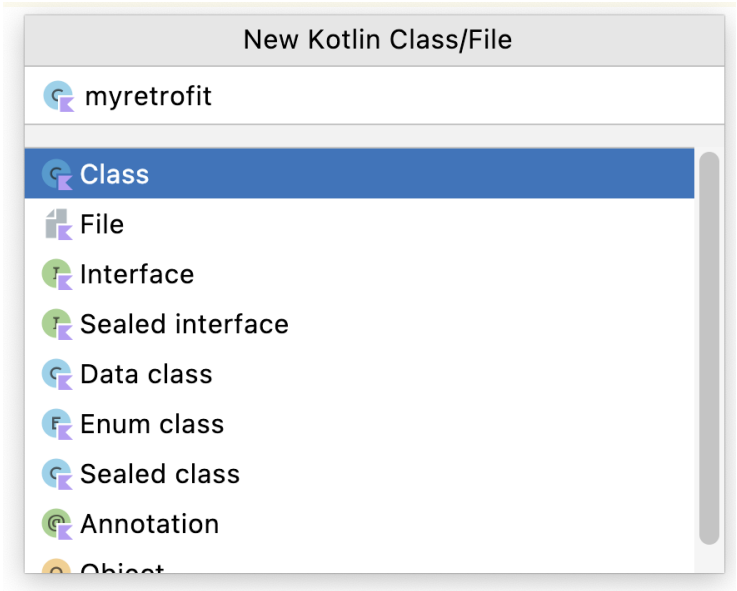


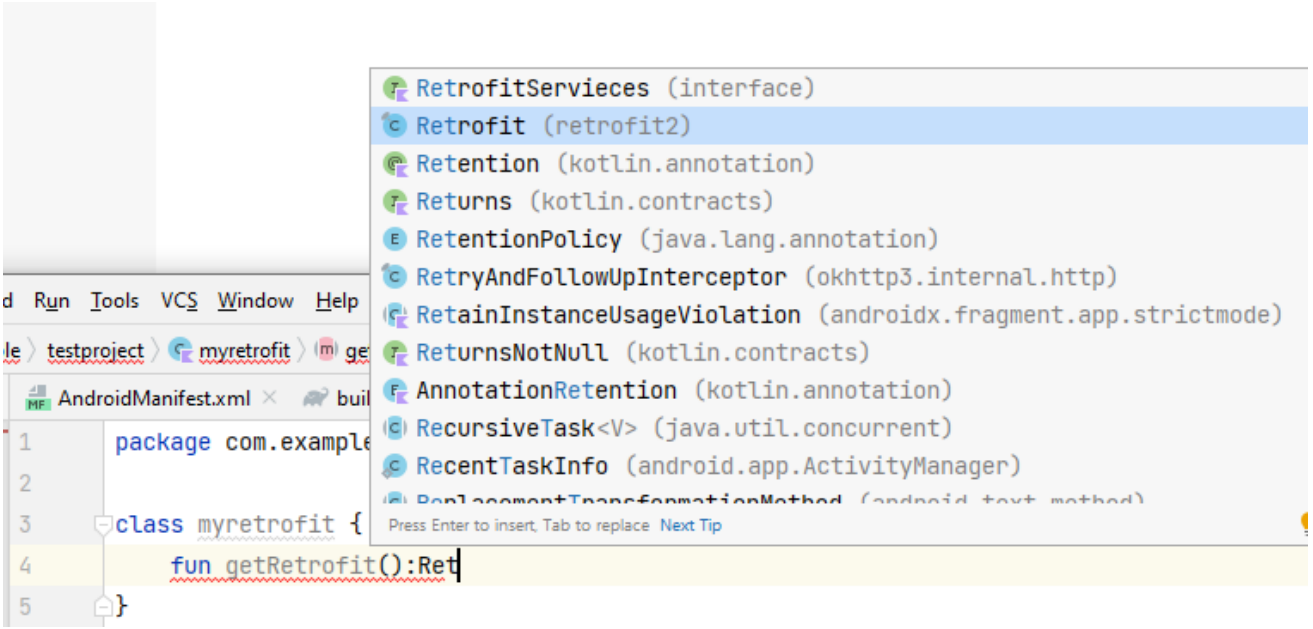
Call<data>, где data - имя дата класса, который вы создаете для преобразования json в pojo

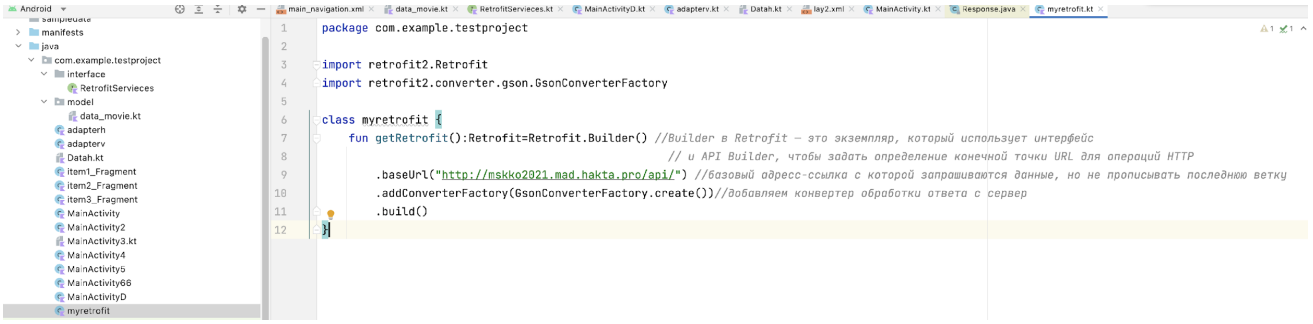


Создайте класс для ретрофита, который в дальнейшем будет использовать в активностях для обработки результатов.









Объект для запроса к серверу создается в простейшем случае следующим образом

```
public static final String BASE_URL = "http://api.example.com/";

Retrofit retrofit = new Retrofit.Builder()
        .baseUrl(BASE_URL)
        .addConverterFactory(GsonConverterFactory.create())
        .build();
```

*В итоге получился объект **Retrofit**, содержащий базовый URL и способность преобразовывать JSON-данные с помощью указанного конвертера Gson. Далее в его методе **create()** указывается класс интерфейса с запросами к серверу, например (в коде методички запись будет отличаться, но смысл останется).*

```
UserService userService = retrofit.create(UserService.class);
```

*После этого будет получен объект **Call** и вызван метод **enqueue()** (для асинхронного вызова) и создан для него **Callback**. Запрос будет выполнен в отдельном потоке, а результат придет в **Callback** в main-потоке.*

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

*Основная часть работы происходит в **onResponse()**, ошибки выводятся в **onFailure()** (неправильный адрес сервера, некорректные формат данных, неправильный формат класса-модели и т.п). HTTP-коды сервера (например, 404) не относятся к ошибкам.*

*Метод **onResponse()** вызывается всегда, даже если запрос был неуспешным. Класс **Response** имеет удобный метод **isSuccessful()** для успешной обработки запроса (коды 200xx).*

Пропишите загрузку фотографий с сервера и цитат в ресайклер с описанием персонажей мультфильма.

Добавьте в проект зависимость для работы с библиотекой GLADE

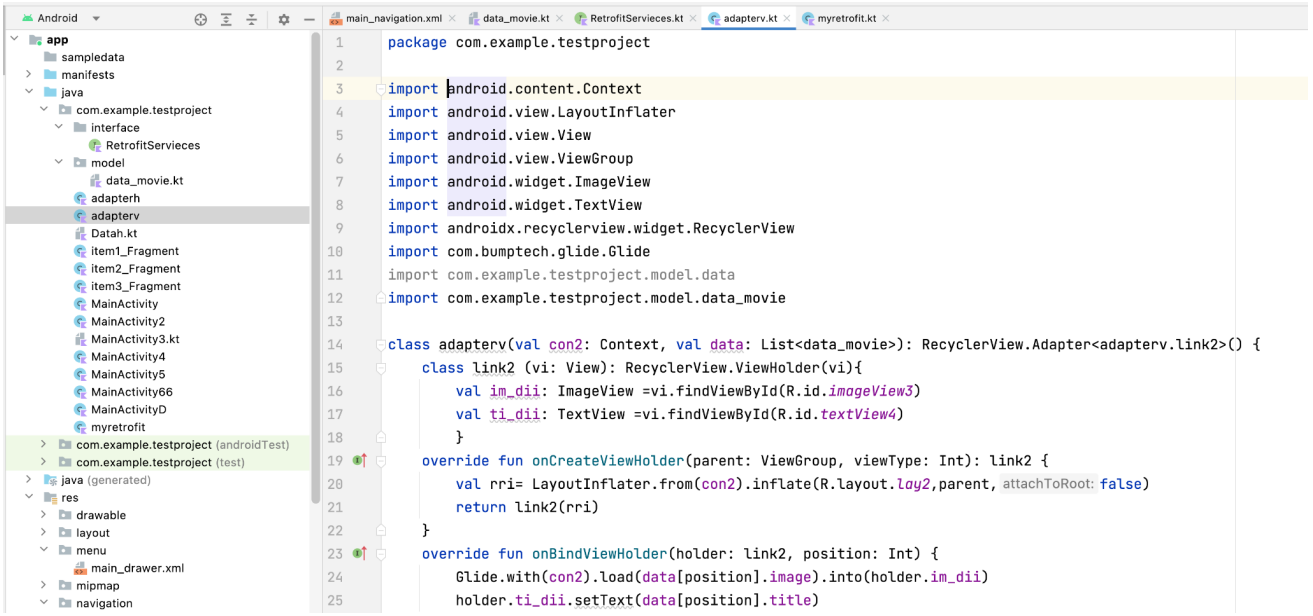
implementation 'com.github.bumptech.glide:glide:4.12.0'

*Библиотека **Glide** (аналог библиотека Picasso) предназначена для асинхронной подгрузки изображений из сети, ресурсов или файловой системы, их кэширования и отображения.*

https://bumptech.github.io/glide/

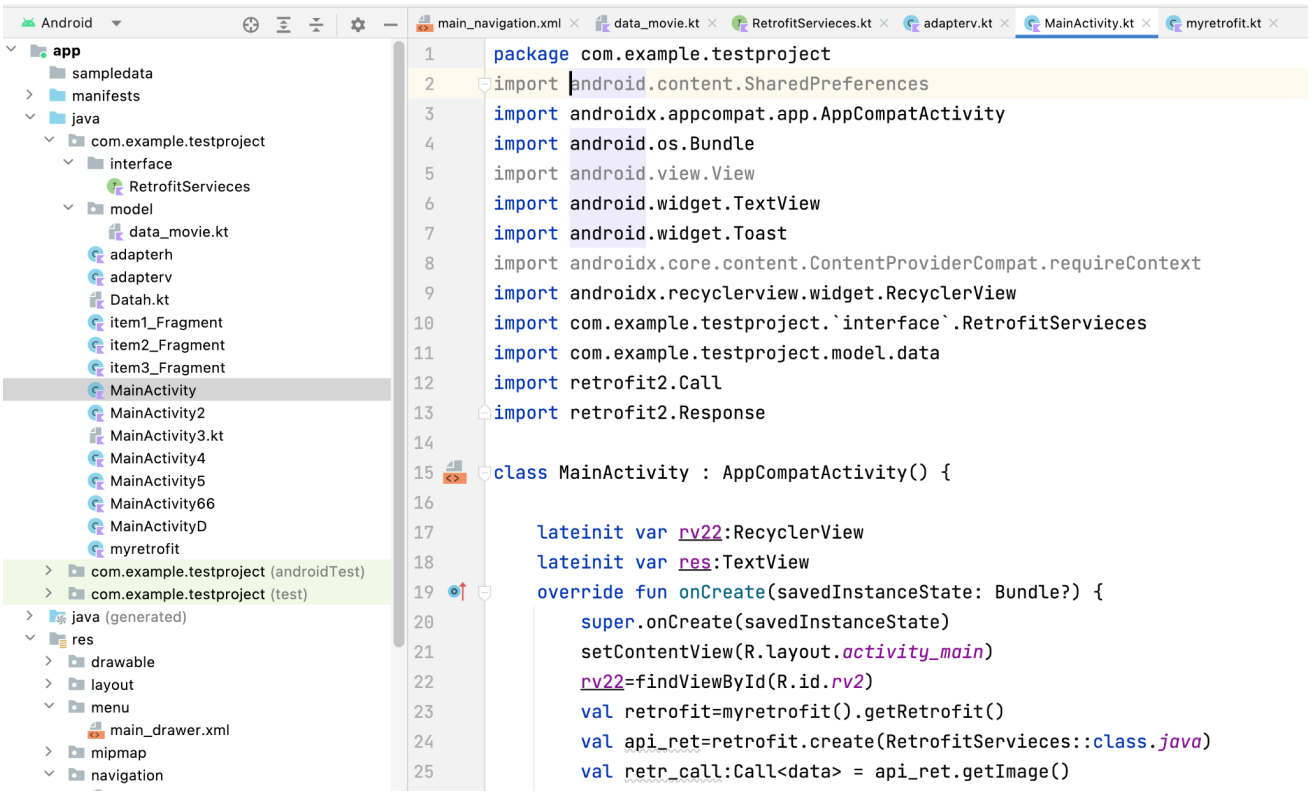
https://github.com/bumptech/glide

Удалите дата класс ресайклера (он больше не нужен, данные будут подгружаться по запросу). Измените класс адаптера для вашего ресайклера (обратите внимание, при автозавершении выбирайте варианты с retrofit2, обратите внимание на имена классов, которые используются):




```
26      }
27      override fun getItemCount(): Int {
28          return data.size
29      }
30  }
```

Подключите возможность обращения к ретрофиту в файле логики активити и подкорректируйте подключение адаптера:



```
        val retr_call:Call<data> = api_ret.getImage()
        retr_call.enqueue(object:retrofit2.Callback<data>
        {
            override fun onResponse(call: Call<data>, response: Response<data>) {
                if(response.isSuccessful)
                {
                    rv22.adapter=response.body()?.let { adapterv(applicationContext,it.data) }
                }
            }
            override fun onFailure(call: Call<data>, t: Throwable) {
                Toast.makeText(applicationContext, t.localizedMessage, Toast.LENGTH_SHORT).show()
            }
        })
    }}
```

Запустите приложение. Если вы все сделали правильно, вместо кадров и описания персонажей вы увидите следующие изображения и текст:


10:21
TEXTVIEW
Спокойным
TEXTVIEW
Расслабленным
TEXTVIEW
Сосредоточенным
TEXTVIEW
Уставшим