1. **To Prevent the app from destroy when rotating the screen we use**

android:configChanges="orientation|screenSize|screenLayout|keyboardHidden"

we write this line inside the app **AndroidManifest.xml** file, inside each **activity** that we don’t want it to get destroyed when rotating.

1. **Alert Dialog: Shorter Way**:

Ex: AlertDialog.Builder(this)  
 .setTitle(“connectionStatus”)  
 .setPositiveButton("RETRY")**{**\_,\_ **->** recreate()**}** .show()

1. **onConfigurationChanged:** Used to **detect the screen rotating** and do a resize for UI element to fix the looks. Used outside the onCreate fun.

Ex: override fun onConfigurationChanged(newConfig: Configuration) {  
 super.onConfigurationChanged(newConfig)  
 if (newConfig.orientation === Configuration.*ORIENTATION\_LANDSCAPE*) {  
 display.setPadding(0,0,24,0)  
 display.*textSize* = 24f  
 }else if (newConfig.orientation===Configuration.*ORIENTATION\_PORTRAIT*){  
 display.setPadding(0,24,24,0)  
 display.*textSize* = 32f  
 }  
 }

1. **To prevent the screen rotating and Set the View to Fixed Orientation** for any activity file we write this line inside AndroidManifest.xml file inside each activity:

android:screenOrientation="portrait"  
android:turnScreenOn="false"

1. **Using the binding method** that we learned in recyclerView, in the **Main Class** to **Bind our variable with UI Elements** and **short our fidViewById**

Ex: override fun onCreate(savedInstanceState:Bundle?) {

super.onCreate(savedInstanceState)

**val binding = ActivityMainBinding.inflate(layoutInflater) 🡺 binding line**

binding.title.text = "Hello"

binding.subtext.text = "Concise, safe code"

binding.button.setOnClickListener { /\* ... \*/ }

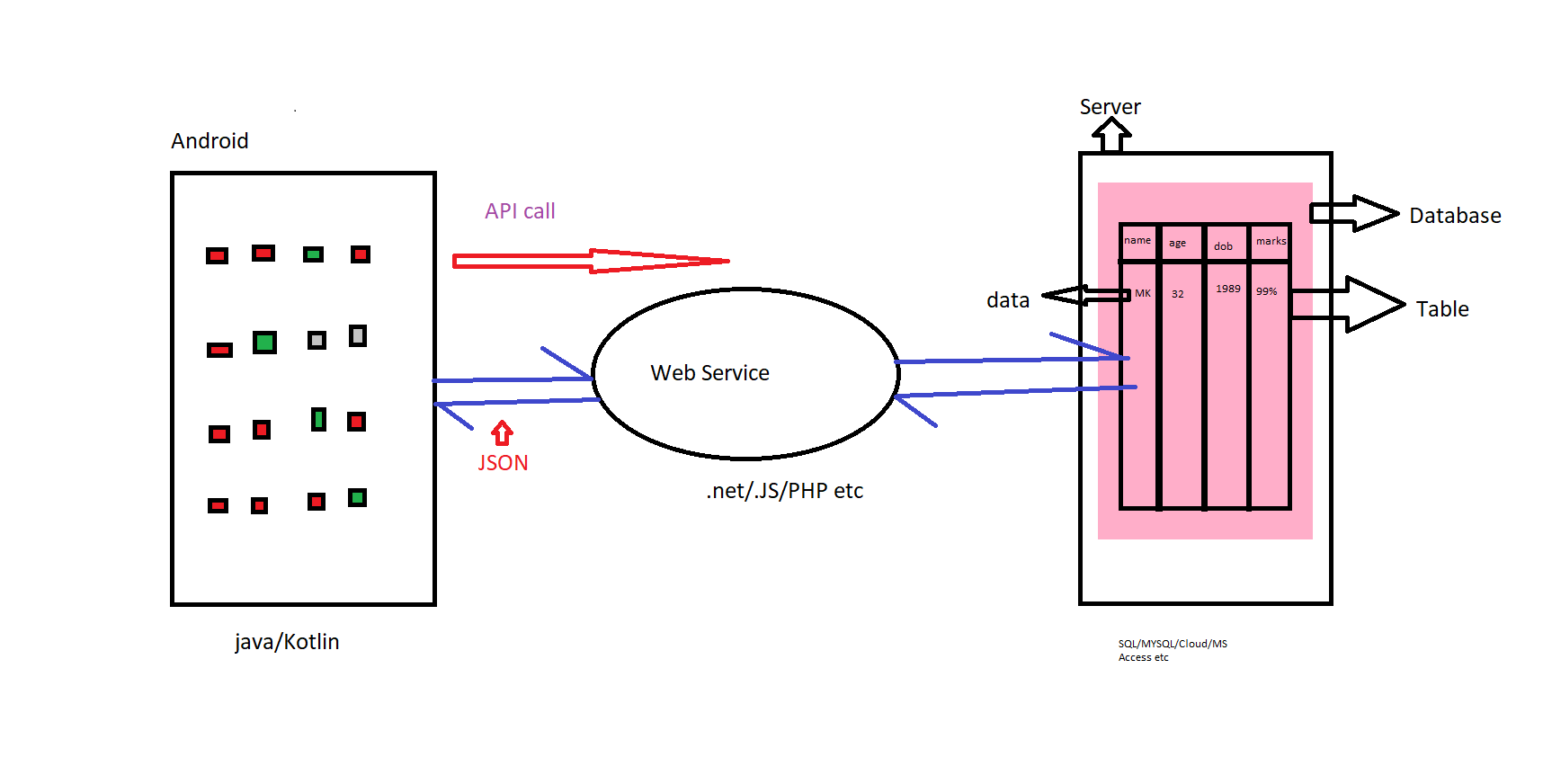
setContentView(binding.root) 🡺 needed to make it the active view on the screen

}

* Don’t forget to fix the **Gradle File**:

Int build.gradle file

android {  
    ...  
    buildFeatures {  
        viewBinding = true  
    }  
}

1. **API and JESON :**

**JSON** stands for Java Script Object Notion (JSON).

It is a data exchanging format between your Webservice and your Android app.

**JSON Syntax:** {“Key”: “Value”} Key will always be a string

Value can be a string, a number, an object, an Array! It can be anything. It is possible to have an object that contains multiple objects and arrays.

We refer to Objects as **JsonObject**

Arrays are called **JsonArrays**

**JsonObjects** are denoted with **Curly braces {}**

**JsonArrays** are denoted with **Curly square brackets []**

**Example JSON format data:**

{“Name”:”CodingDojo”} simple JSON example with value as single string

{ “Age”  : 26 } simple JSON example with value as single number

{“Details” : {“Name”:”CodingDojo”}  } simple JSON example with value as a single JsonObject

{“Users”: [

              {“Name”:”CodingDojo”} ,

              {“Name”:”CodingDojo”} ,

              {“Name”:”CodingDojo”}

              ]

} simple JSON example with value as JsonArray

**API** stands for Application Programming Interface. An API is a URL. if you call it, it is going to give the result in JSON format.

For example, this API <https://fcc-weather-api.glitch.me/api/current?lat=35&lon=139> gives weather data in JSON format.

1. **Add internet permission** in the AndroidManifest file above the <application> tag

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

1. **Retrofit**: In order to work with **Retrofit**, let’s divide the steps in using Retrofit HttpClient.

* Add the dependencies in the dependency section

\*In the build.gradle file (Module level) you will have to add the following decencies:

implementation **'com.google.code.gson:gson:2.8.8'**

implementation **'com.squareup.retrofit2:retrofit:2.9.0'**

implementation **'com.squareup.retrofit2:converter-gson:2.9.0'**

implementation **'com.squareup.okhttp3:logging-interceptor:4.2.2'**

* Understand your result and based on that create a Model class, to hold data

Ex: class Prices{  
//this allows us to access the nested JSON object'eur'to get the price data  
 var eur: Currency? = null  
 class Currency{  
 // here we get individual price data  
 // we don't need serializers  
 var usd: String? = null  
 var aud: String? = null  
 }  
}

Note : **var eur: JsonObject?** = null 🡺 will call all the eur currency (no need to call one by one)

* Declare and initialize the Retrofit instance

Ex: class APIClient {  
 private var retrofit: Retrofit? = null  
 fun getClient(): Retrofit? {  
 retrofit = Retrofit.Builder()  
 .baseUrl("https://cdn.jsdelivr.net/gh/fawazahmed0/currency-api@1/latest/currencies/")  
 .addConverterFactory(GsonConverterFactory.create())  
 .build()  
 return retrofit }  
}

* Start getting data from Retrofit!

Ex: interface APIInterface {  
 @GET("eur.json")  
 fun getPrices(): Call<Prices>  
}

* Use the data in your Android app, in the Main Activity

val apiInterface = APIClient().getClient()?.create(APIInterface::class.*java*)  
// here we use the enqueue callback to make sure that we get the data before we update the Recycler View  
// enqueue gives us async functionality like coroutines   
apiInterface?.getPrices()?.enqueue(object: Callback<Prices>{  
 override fun onResponse(call: Call<Prices>, response: Response<Prices>){  
 try{

// once we get our data, we can update the prices Array List and use it to update the Recycler View  
 prices.add(response.body()!!.eur!!.usd!!)  
 prices.add(response.body()!!.eur!!.aud!!)  
 rvAdapter.notifyDataSetChanged()  
 }catch(e: Exception){ Log.d("MAIN", "ISSUE: $e") }  
 }  
 override fun onFailure(call: Call<Prices>, t: Throwable) {  
 Log.d("MAIN", "Unable to get data") }  
})

1. When we call variable as JsonObject we can use **.keySet()** to **get all keys** variable from JsonObject.

Ex:

val resource: ConvertDetails? = response.body()

val datumList = resource.eur  
if (datumList != null) {  
 val datum = datumList.keySet()

for (value in datum) {  
 listOfCovertsNO.add(datumList.get(value).toString().*toDouble*())  
 //Log.d("TAG","${datumList.get(value).toString().toDouble()}")  
 listOfCoverts.add(value)  
 }  
}

Note: we write the previous code inside the **onResponse** fun.

1. **Coroutines:** Getting all the data in the database will freeze the whole application until it finishes loading the data, such a situation might lead to a crash in the application.

To avoid this issue, **Coroutines** were introduced in Kotlin to produce a new way of writing code that can run in parallel to others. Coroutines can also be used to run heavy code in a way that doesn’t lock the main thread.

A Coroutine runs jobs in the background thread in an efficient way that will solve the time-consuming operations problem in the main thread

Copy the Dependency and paste it in the “build.gradle”:

dependencies {

implementation("org.jetbrains.kotlinx:kotlinx-coroutines-android:1.3.9")

}

To run time-consuming code in the background using Coroutines we need to define the context “Dispatchers” of the background work which can be (IO, Main, Default).

After defining the context, we can start our coroutines by creating its builder (Launch):

CoroutineScope(IO).launch {

}

We also have globalScope which will keep running as long as your application runs, And lifecycleScope which is bound to the Activity or Fragment lifecycle.

The time-consuming code will be written inside the two curly brackets.

Next, we will create a function which will be used and controlled by coroutines:

private suspend fun getResult(){

}

The suspend modifier is added to the function to define that it can be stopped/resumed by the Coroutines and it can only get called inside the Coroutines scope.

If we need to access views from within Coroutines then we need to change the context of the job to be in the main thread by using withContext(Main).

withContext(Main){

}

1. **Lambda** expressions are anonymous functions that can be passed as parameters, stored in variables, or returned as values.

Ex:

data class HighScore(val name: String, val score: Int)

fun main(){

// Lambda

val greeting = { println("Hello") }

greeting()

// Function

fun greetingFun(){

println("Hello")

}

greetingFun()

// Lambda

val personalGreeting = { name:String -> println("Hello $name") }

personalGreeting("Sam")

// Function

fun personalGreetingFun(name: String){

println("Hello $name")

}

personalGreetingFun("Sam")

// Lambda

val returnSum = { a: Int, b: Int -> a + b }

println(returnSum(4, 5))

// Function

fun returnSumFun(a: Int, b: Int): Int{

return a + b

}

println(returnSumFun(4, 5))

val highScores = listOf(

HighScore("Bob", 5000),

HighScore("Sara", 300),

HighScore("Jane", 7200),

HighScore("adfasfds", 900),

HighScore("Fred", 2300)

)

// Lambda

val highestScore = highScores.maxByOrNull { it.score } // it replaces highScores -> highScores.score

println("${highestScore!!.name} wins!")

// Function

fun highestScoreFun(): HighScore{

var hs = 0

var winner = highScores[0]

for(i in highScores){

if(i.score > hs){

hs = i.score

winner = i

}

}

return winner

}

println("${highestScoreFun().name} wins!")

}

1. **Checking the connection** in most of the Apps is crucial because it can give us the opportunity to decide when to start a network call or stop it.

First, we need to use Connectivity Manager which is a part of our Android system.

val cm = this.getSystemService(Context.CONNECTIVITY\_SERVICE) as ConnectivityManager

After creating the Connectivity Manager object we can access the features it holds, one of the features that Connectivity Manager has is returning the info about a connection.

val activeNetwork: NetworkInfo? = cm.activeNetworkInfo

Lastly, we can check the connection by using isConnectedOrConnecting which returns True if it is connected or connecting and false otherwise.

activeNetwork?.isConnectedOrConnecting

if (activeNetwork?.isConnectedOrConnecting == true) { //do something with the network }

else

{ println("Not Connected To The Internet") }

For further information about Connectivity Manager click [here](https://developer.android.com/reference/android/net/ConnectivityManager).

**Note:** Don’t forget to give permissions in Manifests file

<uses-permission android:name="android.permission.ACCESS\_NETWORK\_STATE"/>  
<uses-permission android:name="android.permission.INTERNET"/>

1. Examples of **3rd party libraries**:

|  |  |
| --- | --- |
| [Android PullToRefresh](https://github.com/chrisbanes/Android-PullToRefresh) | [YouTube](https://github.com/PierfrancescoSoffritti/android-youtube-player)  To play youtube videos |
| [Nine Old Android](http://nineoldandroids.com/) | [Picasso](https://square.github.io/picasso/)  To download photos |
| [okHttp](http://square.github.io/okhttp/) | [Glide](https://github.com/bumptech/glide)  To download photos |
| [Volley](http://developer.android.com/training/volley/index.html) |  |
| [Google GSON](https://code.google.com/p/google-gson/) |  |
| [Retrofit](http://square.github.io/retrofit/) |  |
| [Picasso](http://square.github.io/picasso/#download) |  |
| [Universal Image Loader](https://github.com/nostra13/Android-Universal-Image-Loader) |  |
| [Room](https://developer.android.com/training/data-storage/room) |  |

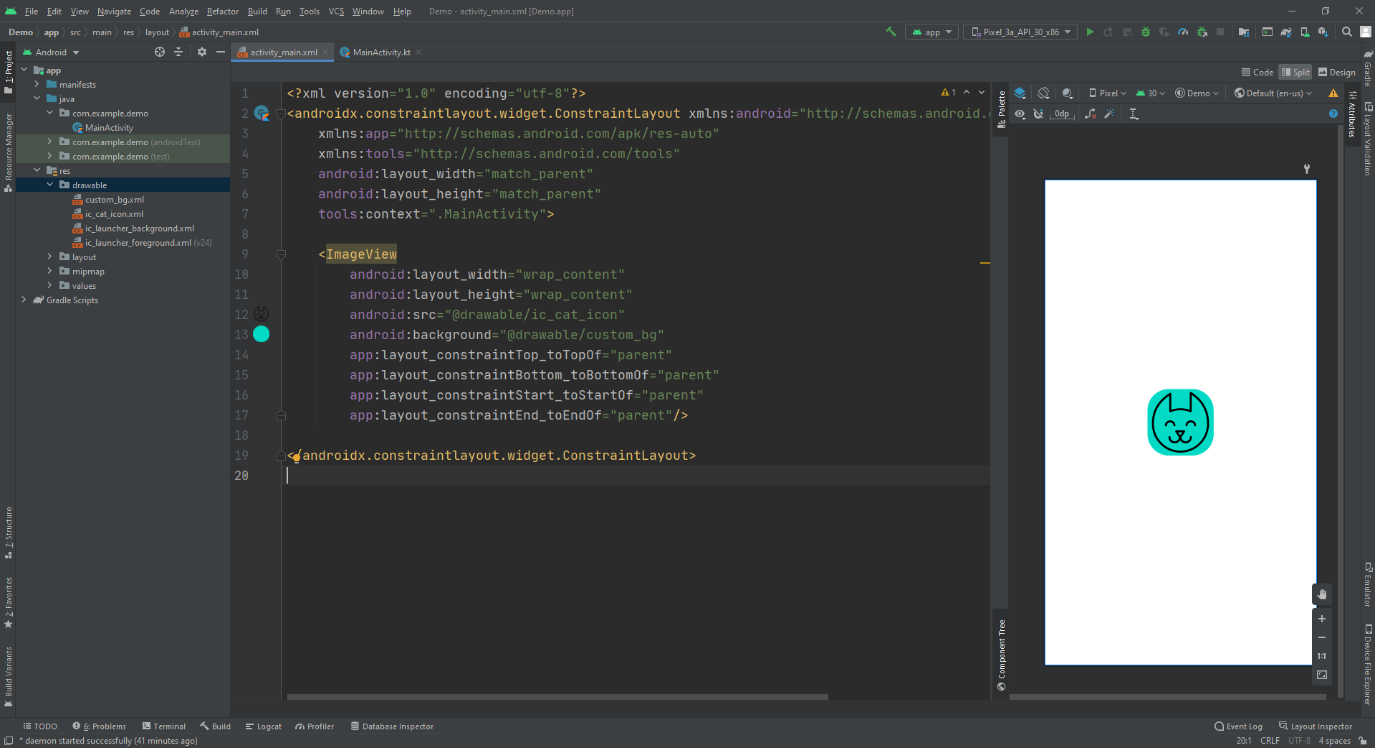
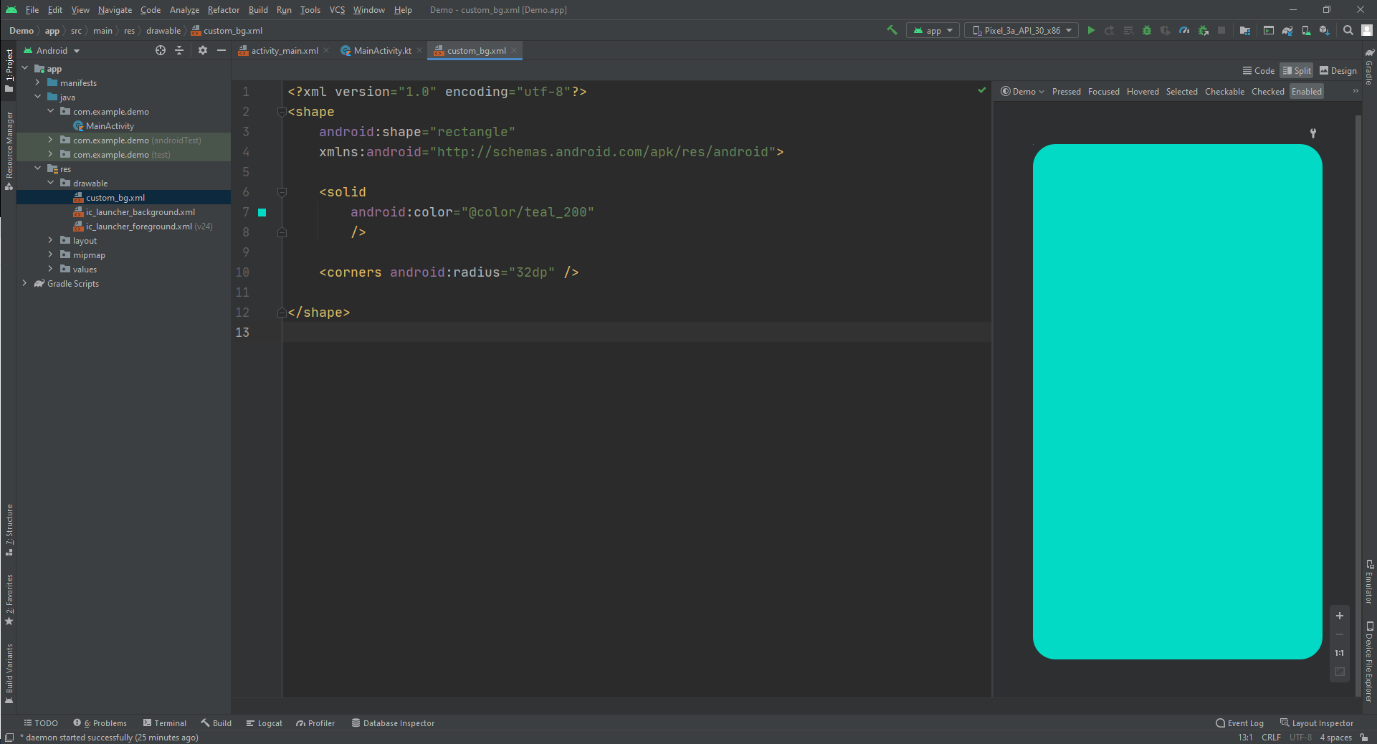
1. To **delete char from string** (**take middle part**) we use **substring** (starting point, end point)

Ex: Var String = “String”.substring(1,”String”.length-1) 🡺 String = “trin”

Ex: var s=”Hello World!”

s=s.substring(0,s.length-1) 🡺 **Deleting last char** 🡺 s=”Hello World”

1. We can use **custom Background** to create **rounded edges background**.

Ex: 

1. **Visibility**: we use **isVisable** to **hide or show out UI elements**.

Ex: EditText.isVisable = false 🡺 will hide the EditText variable

1. To make the **height take spaces by percent %** we write to our UI element in xml file:

app:layout\_constraintHeight\_default=”percent”

app:layout\_constraintHeight\_percent=”0.15”

* + This will make the Height takes only 15% from the screen
  + This helps to control the spaces when rotating

1. You can use **setOnClickListener** to any UI element.

Ex: TextView.seOnClickListener{ //do something }

ImageView.seOnClickListener{ //do something }

LineerLayout.setOnClickListener{ //do something }

1. **Spinner**: is type of UI elements to show data in scroll down list.

Ex: val array = *arrayListOf*("inr", "usd", "aud", "sar", "cny", "jpy")  
 var selected: Int = 0

val spinner = findViewById<Spinner>(R.id.*spinner*)

if (spinner != null) {  
 val adapter = ArrayAdapter(  
 this,  
 android.R.layout.*simple\_spinner\_item*, array  
 )  
 spinner.*adapter* = adapter  
  
 spinner.*onItemSelectedListener* = object :  
 AdapterView.OnItemSelectedListener {  
 override fun onItemSelected(  
 parent: AdapterView<\*>,  
 view: View, position: Int, id: Long  
 ) {  
 selected = position  
 }  
 override fun onNothingSelected(parent: AdapterView<\*>) {  
 // write code to perform some action  
 }  
 }  
}

1. To **check if the String has number** we use **contain**

Ex: for(i in 0..9){  
 if(text.*contains*(i.toString())){  
 return true  
 }  
}

1. To check **if String has Special** **Char** we use **contain**

Ex: val specialCharacters = "!@#$%"  
for(special in specialCharacters){  
 if(text.*contains*(special)){  
 return true  
 }  
}

1. To **check if String has Upper Letter**:

Ex: var letter = 'A'  
while (letter <= 'Z') {  
 for (i in text)  
 if(i == letter){  
 return true  
 }  
 ++letter  
}

1. To make **first letter Upper case** whatever the user enter we use **capitalize()**

Ex: var String = String.capitalize()

**POST Requests**

The two most common API calls we make to communicate with the server are GET and POST.

GET is used to retrieve data from the API, while POST is used to pass some parameters to the server.

We make POST requests with the help of a network library, such as Retrofit or Volley.

We will use Retrofit because it is the most commonly used library.

In our example, we will pretend that we are adding a new book to a library API.

First we need to create the book class:

Then, we add the POST request to our API Interface:

Finally, we make the API call and pass and instance of the Book class:

DiffUtil is a utility class that is present inside [androidx.recyclerview.widget.DiffUtil](https://developer.android.com/reference/kotlin/androidx/recyclerview/widget/DiffUtil" \t "_blank) .

The main purpose of using DiffUtill is to identify the changes made in an existing list.  This allows us to only make changes when there is data that needs updating. This is the way DiffUtil increases the efficiency and performance of a RecyclewView.

Let’s take a real time example, consider you have a list that is used to load data to a RecyclewView.  This list contains over 10,000 student names and their information.

There is a single new student enrolled in your college. In this scenario you will add the new data to the list. But when you are adding the updated list to the RecyclewView again you have to reload all the 10,001 data to the RecyclewView. This is where diffUtil can make a huge difference. Instead updating all existing data, we can simply add the new student without making any changes to the others.

DiffUitil is not always necessary, but it can make a big difference with large data sets.

DiffUtil uses Eugene W. Myers's difference algorithm to calculate the minimal number of updates to the existing list.

Visit the following site for more information: <https://developer.android.com/reference/kotlin/androidx/recyclerview/widget/DiffUtil>

You can show **loading message dialog** to the user when loading something using:

val progressDialog = ProgressDialog(this@MainActivity)  
progressDialog.setMessage("Please wait")  
progressDialog.show()

and when the process finish you can delete it by using:

progressDialog.dismiss()

**val x by lazy { className() }**

this is a way to initialize new class and save it in variable, **what lazy do** is: if the class has init fun (function will run auto when class is initialize), and that variable that saved the class never used this will prevent the init fun from running

**Intent**: already explained in KotlinReference1 but this is Extra:

we can send data between activities using Intent.

Ex: in the MainActivity, here I sent data from 2D array

val intent = Intent(this@MainActivity, ShowRecipe::class.*java*)

intent.putExtra("title", informationList[position][0])  
intent.putExtra("author", informationList[position][1])  
intent.putExtra("ingredients", informationList[position][2])  
intent.putExtra("instructions", informationList[position][3])  
startActivity(intent)

sent to ShowRecipe class, and to get data in ShowRecipe class, and print them in TextView:

private lateinit var bundle: Bundle

//Outside The onCreate

bundle= *intent*.*extras*!!

titleTV.*text*= "Recipe Name: ${bundle.getString("title")!!}"  
authorTV.*text*= "Author Name: ${bundle.getString("author")!!}"  
ingredientsTV.*text*= "Ingredients:\n\n${bundle.getString("ingredients")!!}"  
instructionsTV.*text*=

"Instructions:\n\n${bundle.getString("instructions")!!}"

We can change the **EditText** to be able to take Multi Lines entry from the user:

Ex: in XML file:

android:inputType="textMultiLine"

we can **hide ActionBar**(title bar) from the screen by add this them inside the Activity class that we want to hide the action bar, inside Manifest file:

android:theme="@style/Theme.MaterialComponents.DayNight.NoActionBar"

**Note**: you can create custom them inside the themes.xml file and add to the activity in the Manifest file

Delay code

handler.postDelayed(**{** check = false  
 connectionTV.*text* = " $connectionStatus "  
 **}**, 10000)  
 handler.postDelayed(**{** status = 1  
 changeView()  
 **}**, 12000)  
}

Timer Code

var time = object : CountDownTimer(30000, 1000) {  
  
     override fun onTick(millisUntilFinished: Long) {  
         mTextField.setText("seconds remaining: " + millisUntilFinished / 1000)  
     }  
  
     override fun onFinish() {  
         mTextField.setText("done!")  
     }  
 }

time.start()

Sorting 2D Array, the data in the array:

[ [“22”,”name”,”data1”,”data2”,”data3”], [“3”,”name”,”data1”,”data2”,”data3”] ]

We use: to sort 2D array by name

* + With case insensitive

playList.*sortWith*( *compareBy*(String.*CASE\_INSENSITIVE\_ORDER*,**{it**[1]**}**) )

* + Without case insensitive

playList.*sortBy* **{ it**[1] **}**

to sort 2D Array by id:

playList.*sortBy* **{ it**[0].*toInt*() **}**

use this fun to control the device rotation

override fun onConfigurationChanged(newConfig: Configuration) {  
 super.onConfigurationChanged(newConfig)  
 when {  
 newConfig.orientation === Configuration.*ORIENTATION\_LANDSCAPE* -> {  
 //do something When Horizontal  
  
 }  
 newConfig.orientation=== Configuration.*ORIENTATION\_PORTRAIT* -> {  
 //do something When Vertical  
 }  
 }  
}

to take random element from array without repeating

if (playList.size != numberList.size) {  
 var random = Random.nextInt(0, playList.size)  
 while (true) {  
 if (numberList.contains(random))  
 random = Random.nextInt(0, playList.size)  
 else {  
 numberList.add(random)  
 break  
 }  
 }

}

else{  
 informationTV.*text* ="No More Celebrity to Show!"  
}

API to check the names if trash or good

<https://github.com/ravpacheco/check-name>

to add back button to the title bar we use this code

*supportActionBar*?.setDisplayHomeAsUpEnabled(true)

note: don’t forget to mention the class that the button will send you back to, in the manifest file, inside the activity that we add the back button we add parent activity to back to

android:parentActivityName=".MainActivity"

to replace space or any letter with something else, with Ignore Case we use

“String”.*replace*(" ","%20",true)

Here I replaced space with %20 for URL use

# Parsing XML

So far, we have only used JSON when transmitting data between our application and a server. In this session, we are going to learn how to use XML Parsing.

XML is the second most popular format for sharing data between the server and Android applications after JSON. The XML format is very widely used in websites. For example, news sites, RSS feed sites, and blogs use an XML format to update their content frequently. A large amount of data can be transferred between the server and the Android app in the form of XML Parsing.

Source Code: <https://github.com/AlminPiricDojo/ParsingXML>

to add on click item inside the recycler view (click item not the whole recycler)

we change this class (inside the adapter)

class ItemViewHolder(val binding: InformationsViewBinding, listener: OnItemClickListener ): RecyclerView.ViewHolder(binding.*root*){  
  
 init {  
 itemView.setOnClickListener**{** listener.onItemClick(*adapterPosition*)  
 **}** binding.*apply* **{** AnyUiElementNameYouWantToListen.setOnClickListener**{** Log.d("MyLocation","Adapter")  
 listener.onTextClick(*adapterPosition*)  
 **}  
 }** }  
}

and for the interface

interface OnItemClickListener {  
 fun onItemClick(position: Int)  
 fun onTextClick(position: Int)  
}

and for the main we override the functions

adapter = RVAdaptar(playList)

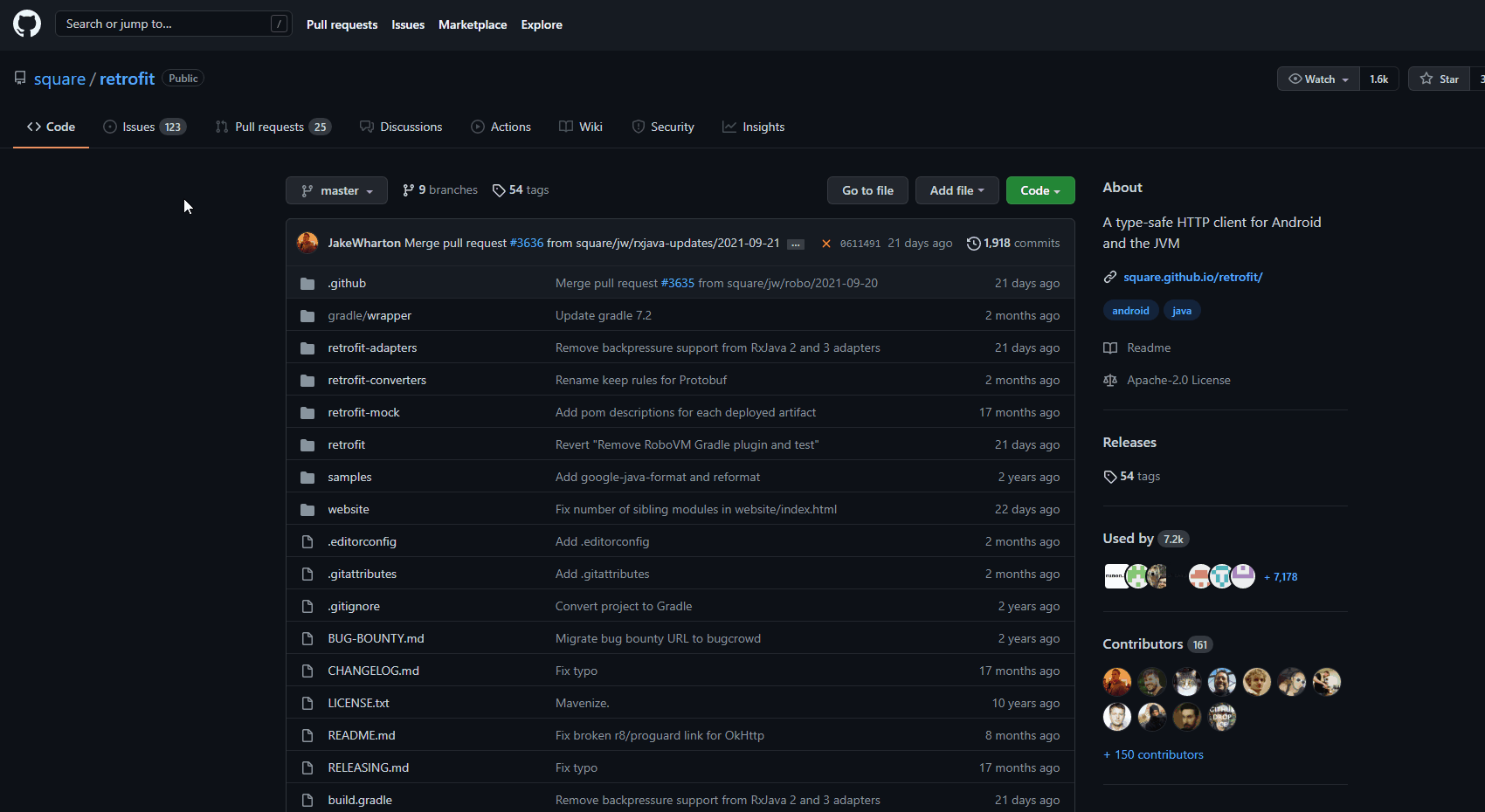
adapter.setOnItemClickListener(object : RVAdaptar.OnItemClickListener {  
 override fun onItemClick(position: Int) {  
 val intent = Intent(this@ShowingData, EditDeleteCelebrity::class.*java*)  
 startActivity(intent)  
 }  
 override fun onTextClick(position: Int) {  
 startActivity(Intent(this@ShowingData, MainActivity::class.*java*))  
 }  
})

# Parse XML Using Retrofit

In the past lesson we covered an XML(RSS) feed and how to parse it using XmlPullParser, this time we will combine it with Retrofit to make the process easier.

Retrofit was used in previous lessons to simplify the connection/parsing of Json files using GsonConverterFactory.

Retrofit supports lots of converters which can work well with it, we can find the converters in the retrofit Github. <https://github.com/square/retrofit>



The used converter for our lesson will be the simpleXml converter because as the name says it is “Simple”, but there are a few points to consider:

1- The library is outdated and deprecated because most of the sites and services have switched to Json structures.

2- The library uses annotations to be able to serialize and match the Kotlin or Java fields with the XML tags.

To add the Simple Xml Converter to the project, we need to add the following dependencies:

implementation ('com.squareup.retrofit2:converter-simplexml:2.2.0')

{

exclude group: 'xpp3', module: 'xpp3'

exclude group: 'stax', module: 'stax-api'

exclude group: 'stax', module: 'stax'

}

In this lesson we will explain some annotations and an example of using them.

* @Root indicates that the class is serialized/deserialized. It represents the element in the XML. With strict set to false, strict parsing is disabled. This tells the parser to not fail and throw an exception if an XML element or attribute is found for which no mapping is provided.
* @Element represents an XML element, The name property signifies the element name. If it is not defined, then the class name is used as the element name.
* @field annotates a Java field.
* @param annotates a Java constructor parameter.

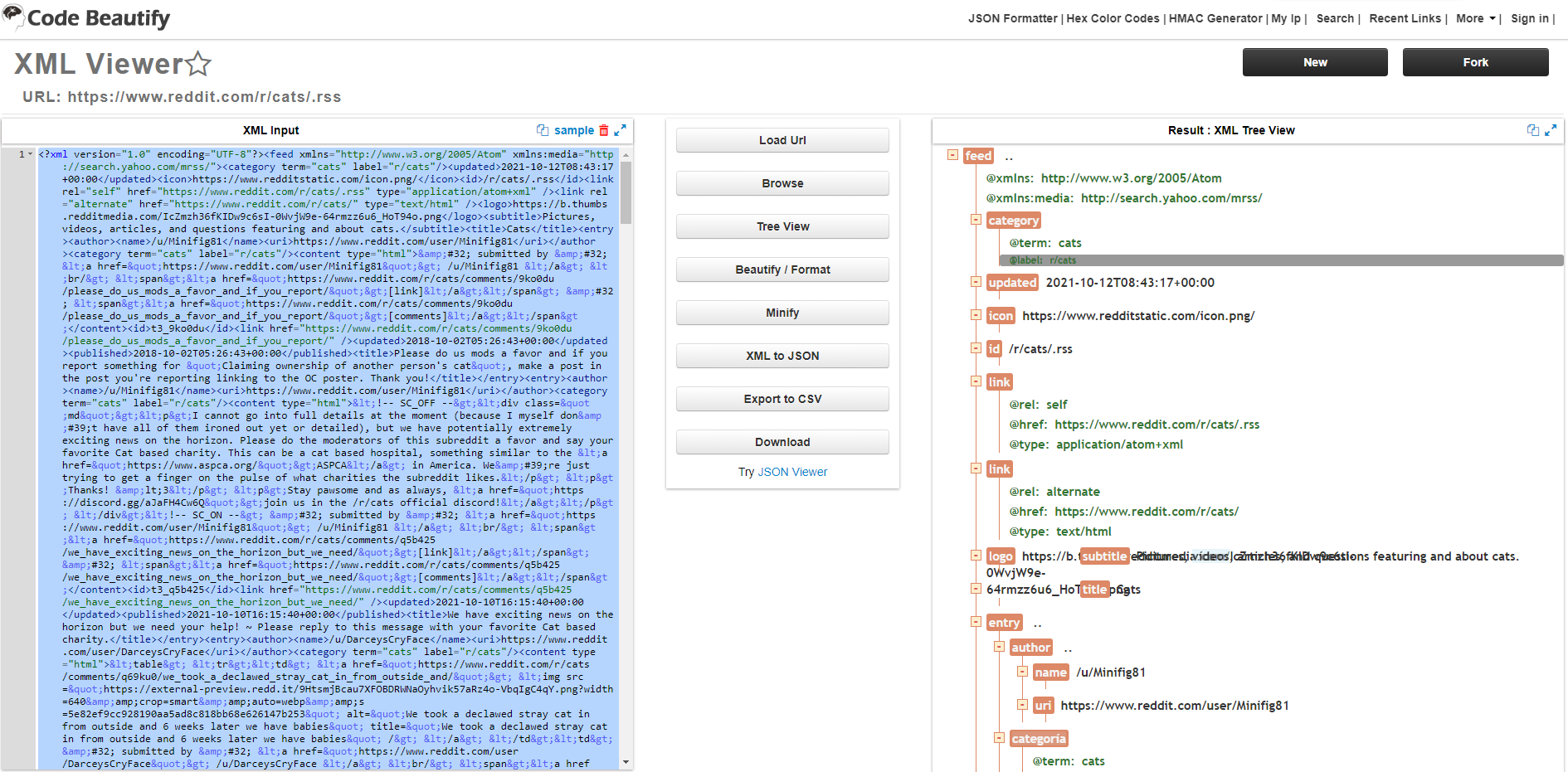
Those annotations are used in most of the Xml classes for serialization purposes.

Let’s jump to an example:

If we go to <https://www.reddit.com/r/cats/.rss> we will see it in a complicated look which will make it hard for us to define the tags in the xml.



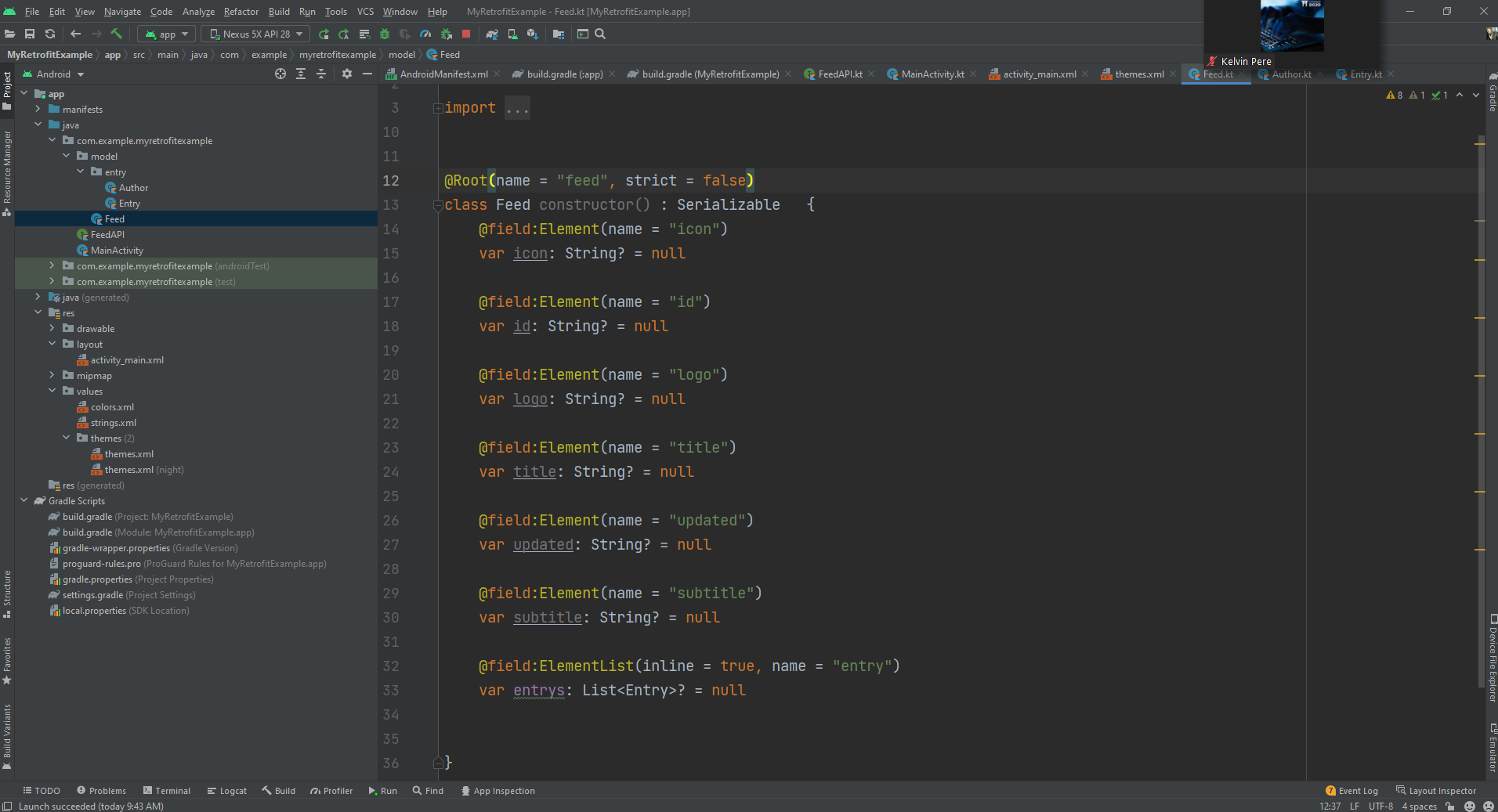
So we will need to make it look better by using <https://codebeautify.org/xmlviewer>



Now we can read the tags on the right side and determine the root which in our case is called 'feed'

And what goes inside of it.

The next step is to create a class for the feed tag.

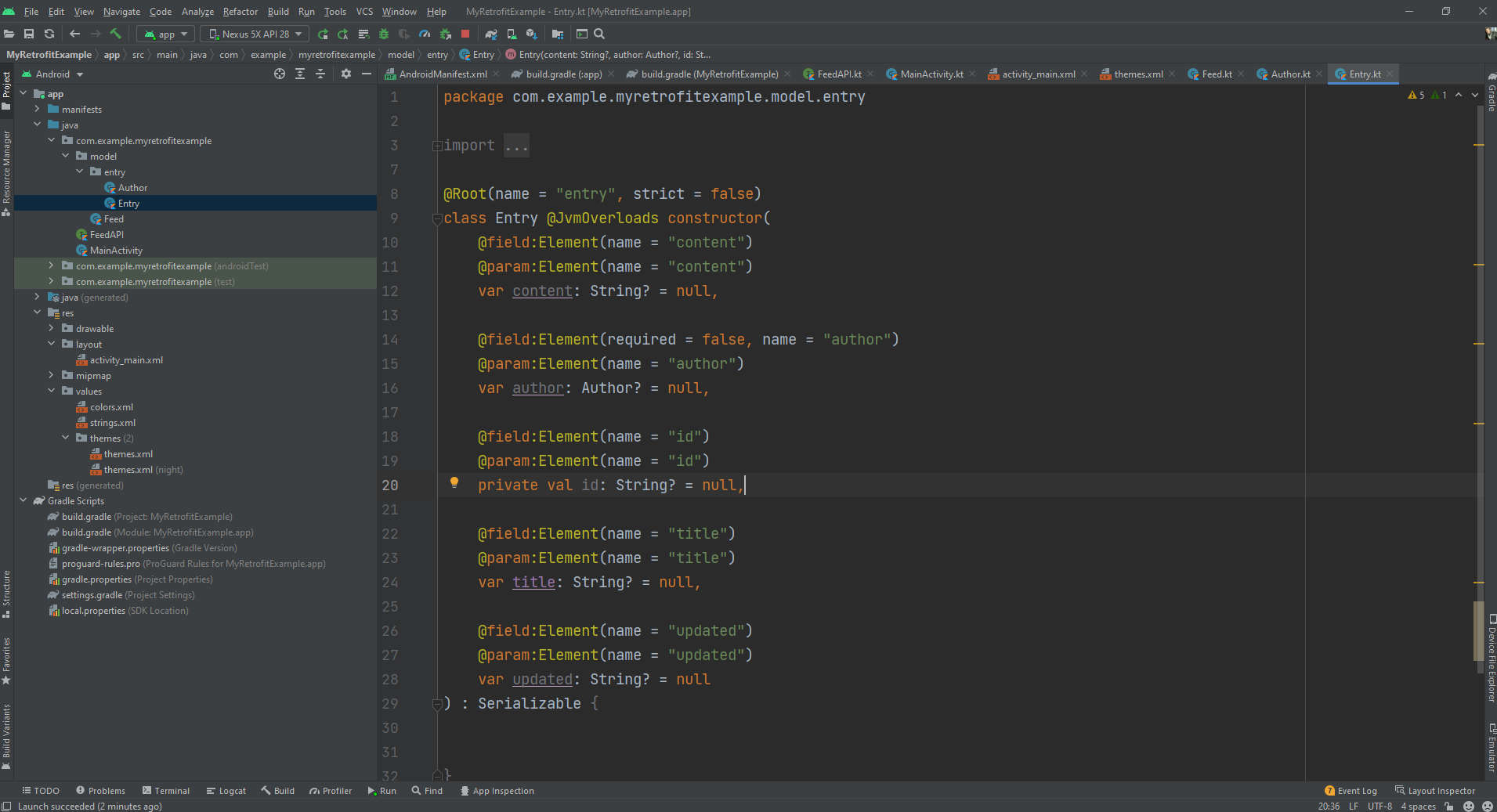


As discussed before, the root defines this class tag and it takes (name and strict) parameters to define if all fields are required or not.

The field:Element and param:Element are required to tell the simple xml that this variable is matched with a field in the tag (feed in our example)

We can also notice that there is ( ElementList ), this element list is defined whenever there is a repeating tag like the entry tag which can be considered a list.

We are interested in entry tags in our example, so we will also create a class for entries.



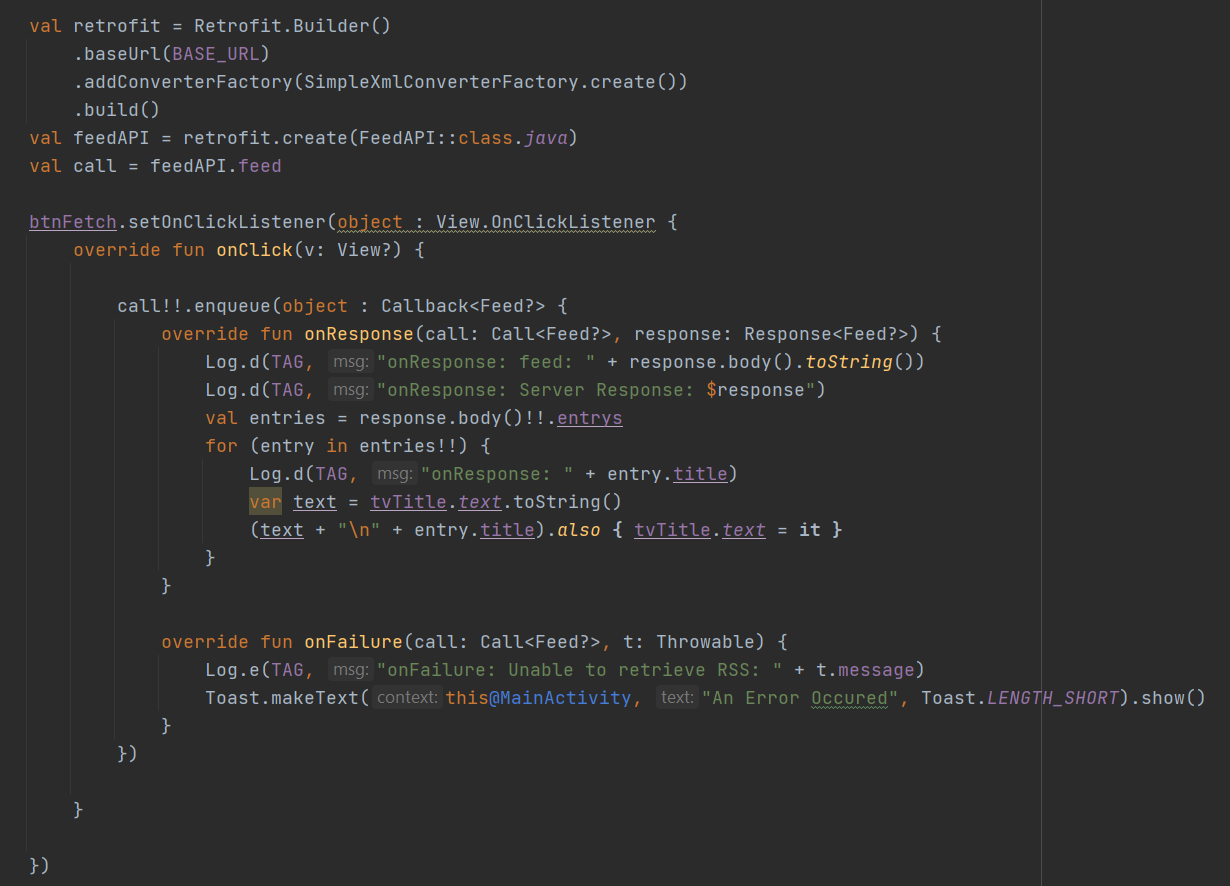
A JvmOverload can be noticed in the entry class, the reason for it is because the simple xml converter library is made to work with empty constructors at first, and Kotlin creates a default constructor with all fields in it, which can throw errors.

The JvmOverload tells Kotlin to override the constructor instead of creating its own.

The retrofit API and the call method will be the same.



And this is what retrofit will look like in our MainActivity (See source code for the full Activity):



For more information on Simple Xml annotations and more tutorials, please go to the this link: <http://simple.sourceforge.net/>

Reddit xml app GitHub link: <https://github.com/Omar-Hedaia-codingdojo/Reddit_Xml_App>