Android Belt Exam 2 Cheat Sheet

**Content**

[**Third-party APIs using retrofit**](#_m50x28au4j0m)

[**GET, POST, PUT, DELETE Requests using retrofit**](#_g1ohsfhgt4tz)

[**Coroutines**](#_r3l0772r7260)

[**Intents**](#_vgb2y7ewn0qx)

[**Companion Objects**](#_les9qgdywr9c)

[**SQLite Local Database**](#_cp7bq6lbo1pj)

[**Room**](#_qsp19brddjw9)

[**ViewModel and LiveData**](#_fi5i54lhun9c)

[**Navigation Component**](#_q7daplbpxypm)

# Third-party APIs using retrofit

Before starting to use the API, you have to understand how it works (API key, requests and requests parameters, response structure…etc). I’ll be using <https://prayertimes.date/api> as an example.

**Setup**

Add retrofit and gson dependencies in module-level build.gradle file:

| implementation 'com.squareup.retrofit2:retrofit:2.9.0' // update to a newer version if available implementation 'com.google.code.gson:gson:2.8.8' implementation 'com.squareup.retrofit2:converter-gson:2.9.0' implementation 'com.squareup.okhttp3:logging-interceptor:4.2.2' |
| --- |

Add internet permission to manifest file:

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

**Implementation**

1. Create a model class based on the JSON response structure, response example:

| {  "code": 200,  "status": "OK",  "results": {  "datetime": [  {  "times": {  "Imsak": "06:23",  "Sunrise": "07:43",  "Fajr": "06:33",  "Dhuhr": "12:34",  "Asr": "14:59",  "Sunset": "17:25",  "Maghrib": "17:44",  "Isha": "18:35",  "Midnight": "23:59"  },  "date": {  "timestamp": 1636070400,  "gregorian": "2021-11-05",  "hijri": "1443-03-30"  }  }  ],  "location": {  "latitude": 48.85661315917969,  "longitude": 2.352221965789795,  "elevation": 36.0,  "city": "Paris",  "country": "France",  "country\_code": "FR",  "timezone": "Europe/Paris",  "local\_offset": 1.0  },  "settings": {  "timeformat": "HH:mm",  "school": "Ithna Ashari",  "juristic": "Shafii",  "highlat": "None",  "fajr\_angle": 12.0,  "isha\_angle": 12.0  }  } } |
| --- |

Corresponds to this model:

| data class ResponseModel ( @SerializedName("code")   val code: Int, @SerializedName("results")   val results: Results, @SerializedName("status")  val status: String )  data class Results( @SerializedName("datetime")  val datetime: List<Datetime>, @SerializedName("location")  val location: Location, @SerializedName("settings")  val settings: Settings )  data class Datetime( @SerializedName("date")  val date: Date, @SerializedName("times")  val times: Times )  data class Date( @SerializedName("gregorian")   val gregorian: String, @SerializedName("hijri")   val hijri: String, @SerializedName("timestamp")   val timestamp: Int ) data class Times( @SerializedName("Asr")   val Asr: String, @SerializedName("Dhuhr")   val Dhuhr: String, @SerializedName("Fajr")   val Fajr: String, @SerializedName("Imsak")   val Imsak: String, @SerializedName("Isha")   val Isha: String, @SerializedName("Maghrib")   val Maghrib: String, @SerializedName("Midnight")   val Midnight: String, @SerializedName("Sunrise")   val Sunrise: String, @SerializedName("Sunset")   val Sunset: String ) data class Location( @SerializedName("city")   val city: String, @SerializedName("country")   val country: String, @SerializedName("country\_code")   val country\_code: String, @SerializedName("elevation")   val elevation: Double, @SerializedName("latitude")   val latitude: Double, @SerializedName("local\_offset")   val local\_offset: Double, @SerializedName("longitude")   val longitude: Double, @SerializedName("timezone")   val timezone: String ) data class Settings( @SerializedName("fajr\_angle")   val fajr\_angle: Double, @SerializedName("highlat")   val highlat: String, @SerializedName("isha\_angle")   val isha\_angle: Double, @SerializedName("juristic")   val juristic: String, @SerializedName("school")   val school: String, @SerializedName("timeformat")   val timeformat: String ) |
| --- |

Tip: you can eliminate properties/classes that you aren't interested in from the model.

2. Create API client class using the base URL of API

| class APIClient {  private var retrofit: Retrofit? = null   fun getClient(): Retrofit? {  val interceptor = HttpLoggingInterceptor()  interceptor.level = HttpLoggingInterceptor.Level.BODY  val client = OkHttpClient.Builder().addInterceptor(interceptor).build()  retrofit = Retrofit.Builder()  .baseUrl("https://api.pray.zone/v2/times/")  .addConverterFactory(GsonConverterFactory.create())  .client(client)  .build()  return retrofit  } } |
| --- |

3. Create API Interface that holds the requests

| interface APIInterface {   @GET("today.json?city=mecca")  fun getTodayPrayerTimes(): Call<ResponseModel?>? } |
| --- |

4. Execute the call in the activity

| private fun getPrayerTimes() {  val apiInterface = APIClient().getClient()?.create(APIInterface::class.java)  val call: Call<ResponseModel?>? = apiInterface!!.getTodayPrayerTimes() // call the method from APIInterface  call?.enqueue(object : Callback<ResponseModel?> {  override fun onResponse(call: Call<ResponseModel?>, response: Response<ResponseModel?>) {  val result = response.body()!!  // do whatever you want with the result (e.g. update UI)  }  override fun onFailure(call: Call<ResponseModel?>, t: Throwable) {  Log.d("The error message", "${t.message}")  call.cancel()  }  })} |
| --- |

# GET, POST, PUT, DELETE Requests using retrofit

There is 4 types of HTTP requests:

* GET: retrieves data.
* POST: inserts data.
* PUT: updates data.
* DELETE: deletes data.

Implement your API Interface according to the requests you want from the API. I’ll be using <https://dojo-recipes.herokuapp.com/celebrities/> as an example since it has all 4 types of requests.

Suppose we have the following model & API Client classes:

| class Celeb : ArrayList<CelebItem>()  data class CelebItem (   @SerializedName("pk")  val pk: Int,   @SerializedName("name")  val name: String,   @SerializedName("taboo1")  val taboo1: String,   @SerializedName("taboo2")  val taboo2: String,   @SerializedName("taboo3")  val taboo3: String ) |
| --- |

| class APIClient {  private var retrofit: Retrofit? = null   fun getClient(): Retrofit? {  val interceptor = HttpLoggingInterceptor()  interceptor.level = HttpLoggingInterceptor.Level.BODY  val client = OkHttpClient.Builder().addInterceptor(interceptor).build()  retrofit = Retrofit.Builder()  .baseUrl("https://dojo-recipes.herokuapp.com")  .addConverterFactory(GsonConverterFactory.create())  .client(client)  .build()  return retrofit  } } |
| --- |

And we want to implement the GET, POST, PUT, and DELETE requests.

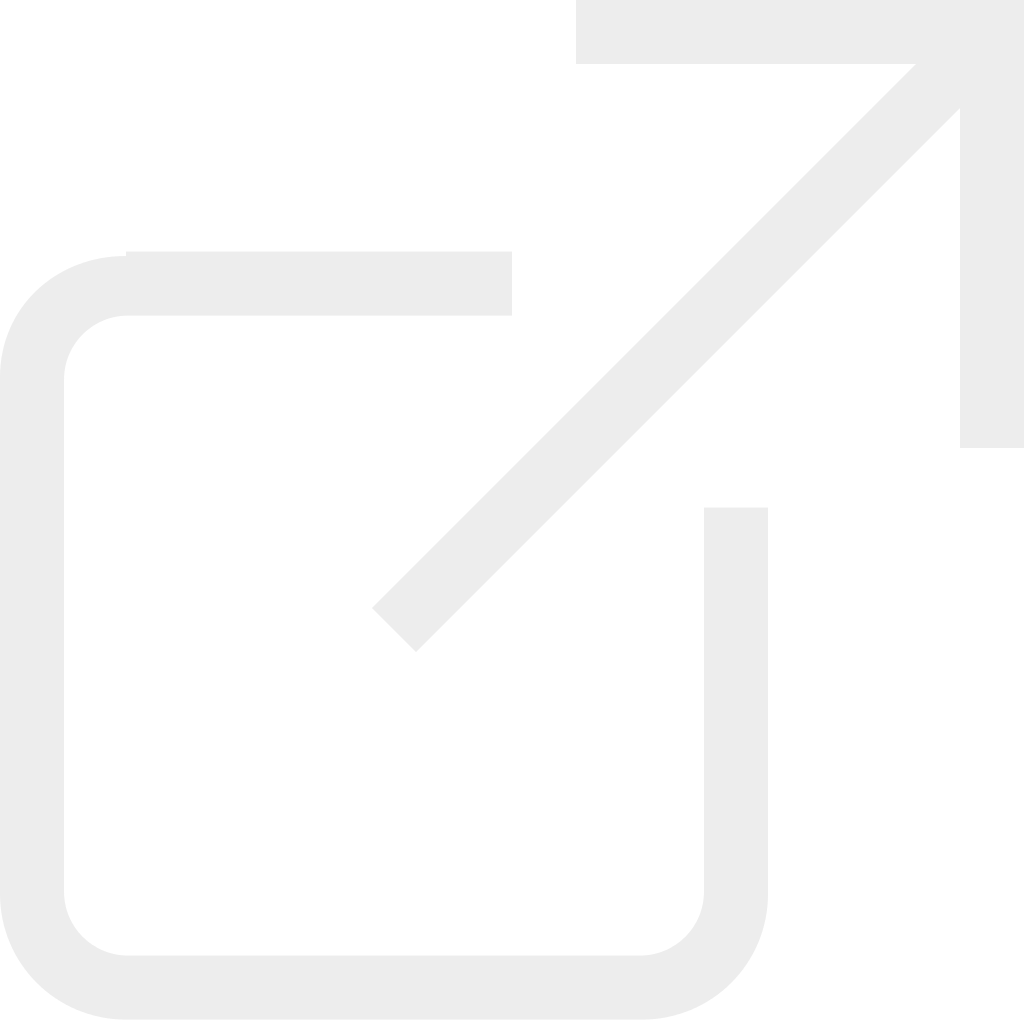
1. Define requests that you want from API docs. In our case, requests are:

* **GET** /celebrities/
* **POST** /celebrities/
* **PUT** /celebrities/{id}
* **DELETE** /celebrities/{id}

2. Implement API Interface

| interface APIInterface {   @POST("/celebrities/")  fun addCeleb(@Body data: CelebItem): Call<CelebItem?>?   @GET("/celebrities/")  fun getCelebs(): Call<Celeb?>?   @PUT("/celebrities/{id}")  fun updateCeleb(@Path("id") id: Int, @Body celebData: CelebItem): Call<CelebItem>   @DELETE("/celebrities/{id}")  fun deleteCeleb(@Path("id") id: Int): Call<Void>? } |
| --- |

Some important annotations:

* @Body: to embed data in the POST and PUT requests.
* @Path(""): to specify a parameter in the request path.
* @Query(""): to specify a query parameter [](https://stackoverflow.com/a/36730379).

# Coroutines

Coroutines help developers to get the advantage of multi-threading by running time-consuming tasks in a background thread that works in parallel with the main thread.

We can execute code in one of 3 dispatchers:

* Default: general purpose dispatcher.
* IO: input/output dispatcher that can be used when fetching data from file, database, web service...etc.
* Main: the main thread which is responsible for the UI.

**Setup**

Add coroutines dependency in module-level build.gradle file:

| implementation "org.jetbrains.kotlinx:kotlinx-coroutines-android:1.4.1" // update to a newer version if available |
| --- |

**Implementation**

Implement the time-consuming task inside coroutine builder, use withContext(*Main*) when updating UI components.

| *CoroutineScope(IO).launch* **{**  val result = timeConsumingTaskResult()  withContext(*Main*) {  someTextView.*text* = result  } **}** |
| --- |

Note: if you want to use retrofit + coroutines, use call?.execute() instead of call?.enqueue().

# Intents

Think of intents as messengers between activities. They allow us to navigate between activities and pass data between them.

There is two types of intents:

* Explicit intents: We use them when we know exactly the “caller” activity.
* Implicit intents: We use them when we have no idea of the “caller” activity, an example was provided in Notifications lesson.

I’ll focus on explicit intents here. Suppose we have 2 activities: MainActivity.kt and AnotherActivity.kt.

To create an intent that navigates and sends data from MainActivity to AnotherActivity:

**MainActivity.kt**

| class MainActivity : AppCompatActivity() {  override fun onCreate(savedInstanceState: Bundle?) {  super.onCreate(savedInstanceState)  setContentView(R.layout.*activity\_main*)  val someEditText = findViewById<EditText>(R.id.*editText*)  val someButton = findViewById<Button>(R.id.*button*)  someButton.setOnClickListener **{**  val intent = Intent(this, AnotherActivity::class.*java*)  intent.putExtra("data", someEditText.*text*.toString())  startActivity(intent)  **}**  } } |
| --- |

**AnotherActivity.kt**

| class AnotherActivity : AppCompatActivity() {  override fun onCreate(savedInstanceState: Bundle?) {  super.onCreate(savedInstanceState)  setContentView(R.layout.*activity\_another*)  val someTextView = findViewById<EditText>(R.id.*textView*)  someTextView.text = *intent*.getStringExtra("data")  } } |
| --- |

# 

# Companion Objects

Companion object is used inside a class to declare properties whose value is shared between class objects (aka static properties). In this way we can access those properties without creating an instance of the class.

Suppose we have this Circle class:



We know that Pi is always 3.14159265 no matter what radius of the circle is. So, using common sense, we will define Pi inside a companion object.

| class Circle(val radius: Float) {  companion object {  val PI = 3.14159265f  }  fun computeArea() = PI \* radius \* radius } |
| --- |

And to access that PI from outside the class (from MainActivity for example)

| class MainActivity : AppCompatActivity() {  override fun onCreate(savedInstanceState: Bundle?) {  super.onCreate(savedInstanceState)  setContentView(R.layout.*activity\_main*)  val pi = Circle.PI // note that we didn’t create an instance of circle  } } |
| --- |

# 

# SQLite Local Database

SQLite is a local database management system and it’s Android Studio built-in. It allows the developer to make the main four operations:

* **C**reate
* **R**ead
* **U**pdate
* **D**elete

or **CRUD**.

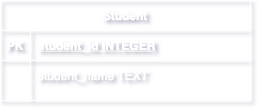
To make a database for our app:

1. Construct Database Helper class

2. Create necessary table(s)

3. Add necessary CRUD operations

Let’s take this simple database schema as an example:



Constructing Database Helper class & creating Student table

| class DBHelper(context: Context): SQLiteOpenHelper(context, "students.db", null, 1) {   private var SQLiteDatabase: SQLiteDatabase = writableDatabase   override fun onCreate(db: SQLiteDatabase?) {  // here we create database table(s)  db?.execSQL("CREATE TABLE Student (student\_id INTEGER, student\_name TEXT)")  }   override fun onUpgrade(db: SQLiteDatabase?, oldVersion: Int, newVersion: Int) {  // onUpgrade is executed when the database version increases, out of our scope :D  *TODO("Not yet implemented")*  }  } |
| --- |

# Room

# ViewModel and LiveData

# 

# Navigation Component