Mobile Application Development

Produced by

David Drohan (ddrohan@wit.ie)

Department of Computing & Mathematics
Waterford Institute of Technology
http://www.wit.ie

Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

Networking with Retrofit

Agenda

- Networking Overview
- Annotations and Classes
- Revisit interfaces
- How to integrate Retrofit into an Android App (Donation)

Overview

- In a nutshell, android networking (or any networking) works with the following:
  - **Request** — Make an HTTP request to an URL (called an endpoint) with proper headers (generally with Authorisation Key if required)
  - **Response** — The Request will return a response which can be error or success. In the case of success, the response will contain the contents of the endpoint (generally in JSON format)
  - **Parse & Store** — parse this JSON and get the required values and store them in our data class

Overview

- In Android, we use
  - **Okhttp** — for creating an HTTP request with all the proper headers
  - **Retrofit** — for making the request
  - **Moshi** / **GSON** — for parsing the JSON data
  - **Kotlin Coroutines** — for making non-blocking (main thread) network requests
  - **Picasso** / **Glide** — for downloading an image from the internet and setting it into an ImageView

Retrofit

A type-safe **HTTP client** for Android and Java

Introduction

Retrofit turns your HTTP API into a Java interface.

```
public interface GitHubService {
  @GET("users/{user}/repos")
  Call<List<Repo>> listRepos(@Path("user") String user);
}
```

The `Retrofit` class generates an implementation of the `GitHubService` interface.

```
Retrofit retrofit = new Retrofit.Builder()
    .baseUrl("https://api.github.com/")
    .build();

GitHubService service = retrofit.create(GitHubService.class);
```

Each `Call` from the created `GitHubService` can make a synchronous or asynchronous HTTP request to the remote webserver.

```
Call<List<Repo>> repos = service.listRepos("octocat");
```

What is it?

- Retrofit is a Java Library that turns your REST API into a Java/Kotlin interface
- Simplifies HTTP communication through declarative & type-safe interfaces
- Developed by Square (Jake Wharton)
- Retrofit is one of the most popular HTTP Client Library for Android as a result of its simplicity and its great performance compared to the others (next slide)
- Retrofit makes use of OkHttp (from the same developer) to handle network requests

Retrofit Performance Analysis

	One Discussion	Dashboard (7 requests)	25 Discussions
AsyncTask	941 ms	4,539 ms	13,957 ms
Volley	560 ms	2,202 ms	4,275 ms
Retrofit	312 ms	889 ms	1,059 ms

Why Use it?

- Developing your own type-safe HTTP library to interface with a REST API can be a burden: you have to handle many functionalities such as making connections, caching, retrying failed requests, threading, response parsing, error handling, and more
- Retrofit, on the other hand, is very well planned, documented, and tested—a battle-tested library that will save you a lot of precious time and headache

The Basics

- Retrofit2 is a flexible library that uses annotated interfaces to create REST calls.
- To get started, let's look at our Donation example that makes a GET request for Donations.
- Here's the DonationModel class we'll be using:
- Much simpler if field names matches server model (but doesn't have to, see later)

```
@Parcelize
data class DonationModel(
    var _id: String = "N/A",
    val paymenttype: String = "N/A",
    val amount: Int = 0,
    val message: String = "a message")
                          : Parcelable
```

The Service interface

- Once we've defined the class, we can make a service interface to handle our API. A **GET** request to load all Donations could look something like this:

```
interface DonationService {
    @GET("/donations")
    fun getall(): Call<List<DonationModel>>
}
```

- Note that the @GET annotation takes the endpoint we wish to access. As you can see, an implementation of this interface will return a **Call** object containing a list of Donations.

The Service interface

- We also need to build a proxy service with the appropriate Gson parsers
- 'Wrap it up' inside a singleton **`create()`** method

```
interface DonationService {
    @GET("/donations")
    fun getall(): Call<List<DonationModel>>

    companion object {

        val serviceURL = "https://donationweb-hdip-server.herokuapp.com"

        fun create() : DonationService {

            val gson = GsonBuilder().create()

            val okHttpClient = OkHttpClient.Builder()
                .connectTimeout(30, TimeUnit.SECONDS)
                .writeTimeout(30, TimeUnit.SECONDS)
                .readTimeout(30, TimeUnit.SECONDS)
                .build()

            val retrofit = Retrofit.Builder()
                .baseUrl(serviceURL)
                .addConverterFactory(GsonConverterFactory.create(gson))
                .client(okHttpClient)
                .build()
            return retrofit.create(DonationService::class.java)
        }
    }
}
```

`Gson` for converting our JSON

`OkHttpClient` for communication timeouts (optional)

`Retrofit.Builder` to create an instance of o ur interface

Retrofit 2.0 introduced a new **Call** class (it also exists in OkHttp package).

Call

- Models a single request/response pair
- Separates request creation from response handling
- Each instance can only be used once...
- ...instances can be cloned
- Supports both synchronous and asynchronous execution.
- Can be (actually) canceled

(screen is taken from Jack Whartons presentation during Droidcon NYC 2015, available here)

Calling the API

- So how do we use this interface to make requests to the API?
- Use Retrofit2 to create an implementation of the above interface, and then call the desired method.
- Retrofit2 supports a number of converters used to map Java/Kotlin objects to the data format your server expects (JSON, XML, etc). We'll be using the Gson converter.

- 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`
- Scalars (primitives, boxed, and String): `com.squareup.retrofit2:converter-scalars`

DonationApp - `onCreate()`

- Create an instance of the proxy service 'donationService'

```
class DonationApp : Application(), AnkoLogger {

    lateinit var donationService: DonationService
    var donations = ArrayList<DonationModel>()

    override fun onCreate() {
        super.onCreate()
        info("Donation App started")
        donationService = DonationService.create()
        info("Donation Service Created")
    }
}
```

Calling the API

- Implement the necessary interface

Note the **Callback** interface

```
class ReportFragment : Fragment(), AnkoLogger,
                       Callback<List<DonationModel>> {
```

- and Callback objects

Called inside **onResume()**

```
fun getAllDonations() {
    showLoader(loader, "Downloading the Donations List")
    var callGetAll = app.donationService.getall()
    callGetAll.enqueue(this)
}
```

enqueue() allows for asynchronous callback to our service

Calling the API

Note the returned **response** on a successful call

- Override the Callback methods

```
override fun onResponse(call: Call<List<DonationModel>>,
                        response: Response<List<DonationModel>>) {
    serviceAvailableMessage(activity!!)
    info("Retrofit JSON = ${response.body()}")
    app.donations = response.body() as ArrayList<DonationModel>
    root.recyclerView.adapter = DonationAdapter(app.donations)
    root.recyclerView.adapter?.notifyDataSetChanged()
    checkSwipeRefresh()
    hideLoader(loader)
}
```

Calling the API

Note the returned **t** on a failed call

- Override the Callback methods

```
override fun onFailure(call: Call<List<DonationModel>>,
                       t: Throwable) {
    info("Retrofit Error : $t.message")
    serviceUnavailableMessage(activity!!)
    checkSwipeRefresh()
    hideLoader(loader)
}
```

Calling the API – onResponse()

- Triggered on a successful call to the API
- Takes 2 parameters
  - The Call object
  - The expected Response object
- Converted JSON result stored in response.body()

```
override fun onResponse(call: Call<List<DonationModel>>, response: Response<List<DonationModel>>) {
    serviceAvailableMessage(activity!!)
    info("Retrofit JSON = ${response.body()}")
    app.donations = response.body() as ArrayList<DonationModel>
    root.recyclerView.adapter = DonationAdapter(app.donations)
    root.recyclerView.adapter?.notifyDataSetChanged()
    checkSwipeRefresh()
    hideLoader(loader)
}
```

Calling the API – `onFailure()`

- Triggered on an unsuccessful call to the API
- Takes 2 parameters
  - The Call object
  - A Throwable object containing error info
- Should inform user of what's happened

```
override fun onFailure(call: Call<List<DonationModel>>, t: Throwable) {
    info("Retrofit Error : $t.message")
    serviceUnavailableMessage(activity!!)
    checkSwipeRefresh()
    hideLoader(loader)
}
```

Calling the API – Helper methods

- 2 helper methods for user feedback

```
fun serviceUnavailableMessage(activity: FragmentActivity) {
    Toast.makeText(activity,
        "Donation Service Unavailable. Try again later",
        Toast.LENGTH_LONG
    ).show()
}

fun serviceAvailableMessage(activity: FragmentActivity) {
    Toast.makeText(activity,
        "Donation Contacted Successfully",
        Toast.LENGTH_LONG
    ).show()
}
```

Beyond GET – other types of Calls

- Retrofit2 is not limited to GET requests. You may specify other REST methods using the appropriate annotations (such as @POST, @PUT and @DELETE).
- Here's another version of our **DonationService** interface

```
interface DonationService {
    @GET("/donations")
    fun getall(): Call<List<DonationModel>>

    @GET("/donations/{id}")
    fun get(@Path("id") id: String): Call<DonationModel>

    @DELETE("/donations/{id}")
    fun delete(@Path("id") id: String): Call<DonationWrapper>

    @POST("/donations")
    fun post(@Body donation: DonationModel): Call<DonationWrapper>

    @PUT("/donations/{id}")
    fun put(@Path("id") id: String,
            @Body donation: DonationModel
    ): Call<DonationWrapper>
```

Headers

- If you wish to add headers to your calls, you can annotate your method or arguments to indicate this.
- For instance, if we wanted to add a content type and a user's authentication token, we could do something like this:

```
@POST("/donations")
@Headers("Content-Type: application/json")
fun post(@Body donation: DonationModel,
         @Header("Authorization") token : String)
        : Call<DonationWrapper>
```

Bridging the Gap Between Your Code & Your API

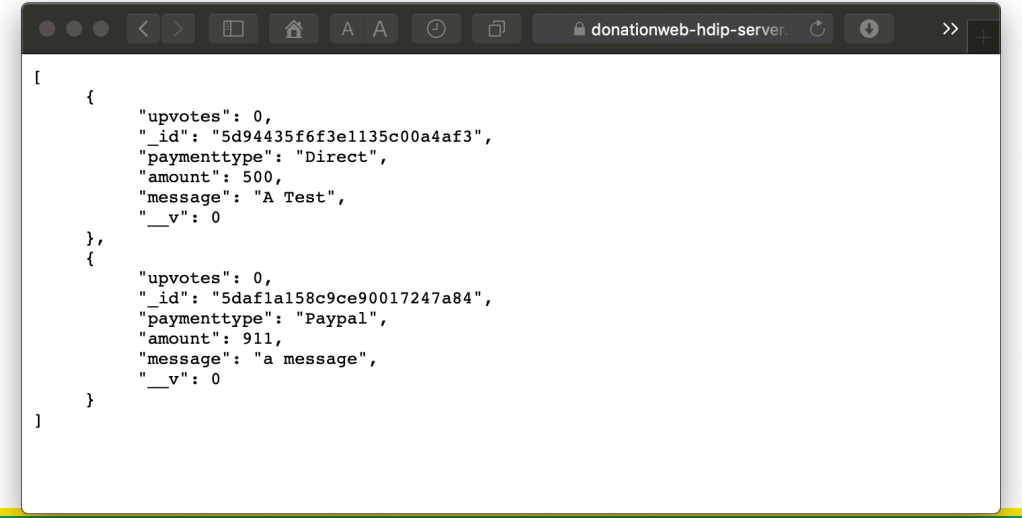
Variable Names

- In the previous examples, we assumed that there was an exact mapping of instance variable names between the Donation class and the server. This will often not be the case, especially if your server uses a different spelling convention than your Android app.
- For example, if you use a Rails server, you will probably be returning data using `snake_case`, while your Java/Kotlin probably uses `camelCase`. If we add a `dateCreated` to the Donation class, we may be receiving it from the server as `date_created`.
- To create this mapping, use the @SerializedName annotation on the instance variable. For example:

```
@SerializedName("date_created")
var dateCreated : Date
```

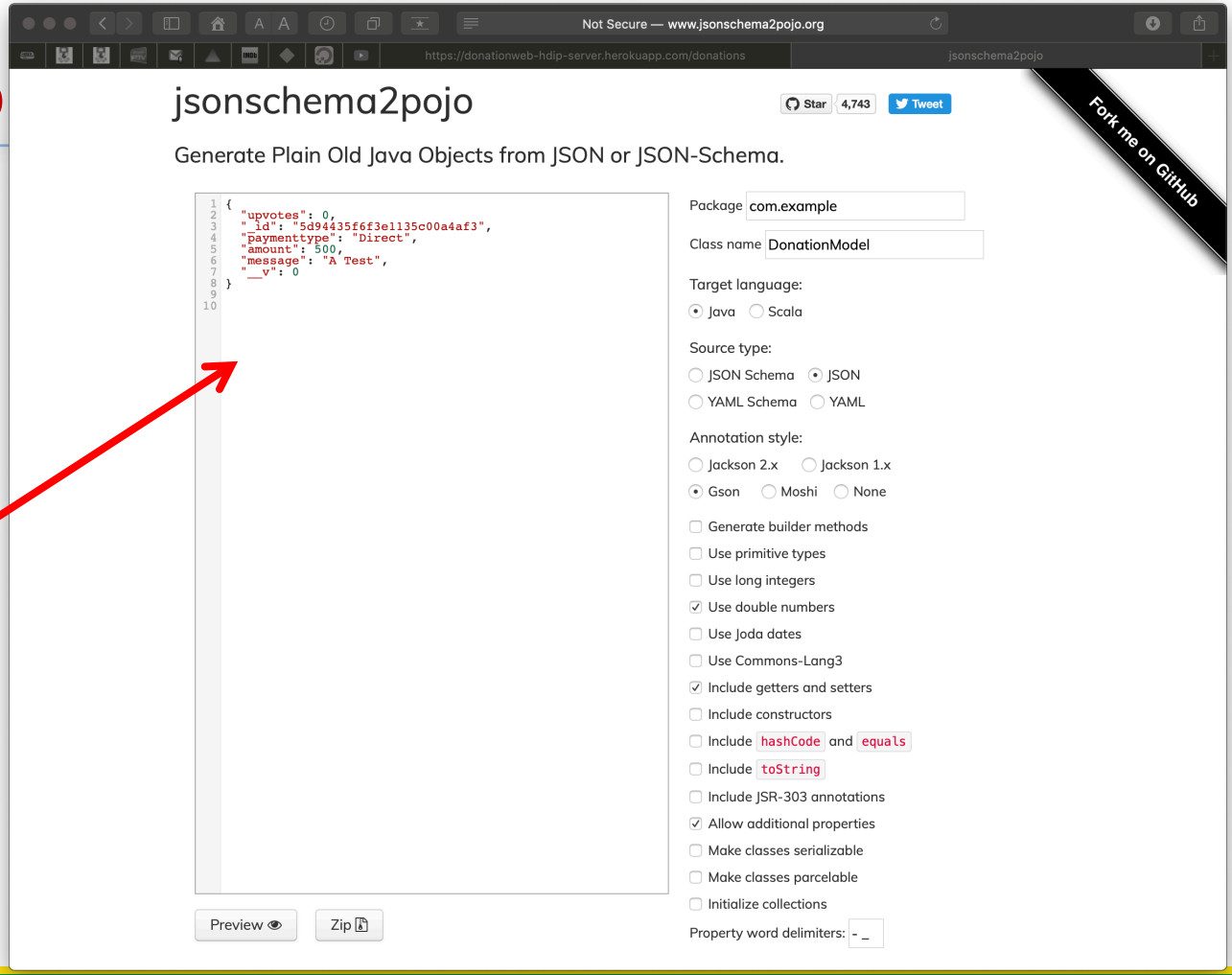
Bridging the Gap Between Your Code & Your API

- You can also create your models automatically from your JSON response data by leveraging a useful tool: jsonschema2pojo - http://www.jsonschema2pojo.org
- Grab your JSON string, visit the above link and paste it in, like so



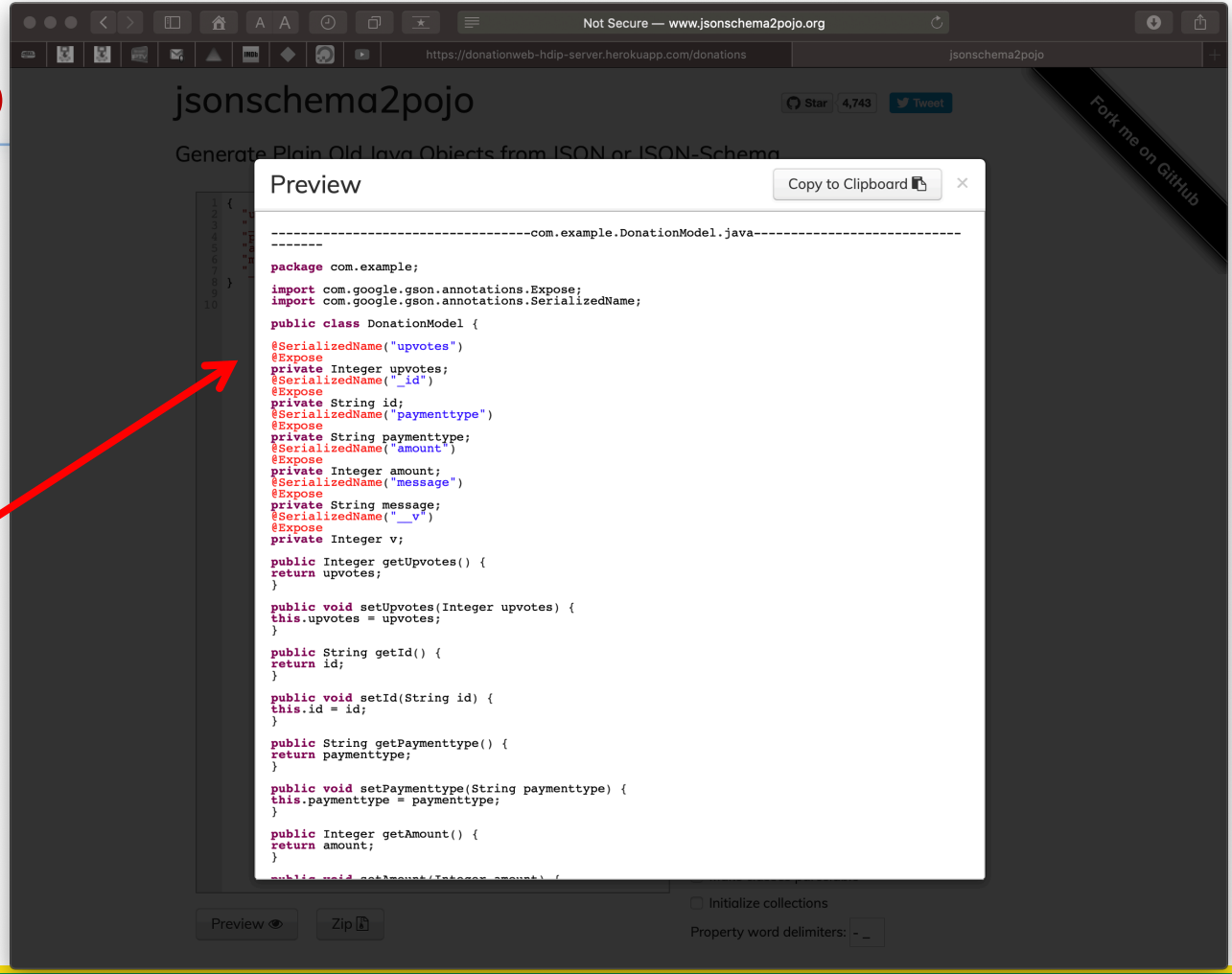
Bridging the Gap Between Your Code & Your API

Your JSON goes here



Bridging the Gap Between Your Code & Your API

Your annotated class



Bridging the Gap Between Your Code & Your API

Date Formats

- Another potential disconnect between the app and the server is the way they represent date objects.
- For instance, Rails may send a date to your app in the format "`yyyy-MM-dd'T'HH:mm:ss`" which Gson will not be able to convert to a Java/Kotlin **Date**. We have to explicitly tell the converter how to perform this conversion. In this case, we can alter the Gson builder call to look like this:

```
val gson = GsonBuilder()
                .setDateFormat("yyyy-MM-dd'T'HH:mm:ss")
                .create();
```

Full List of Retrofit Annotations

- @POST
- @PUT
- @GET
- @DELETE
- @Header
- @PATCH
- @Path
- @Query
- @Body

References

Sources:

https://square.github.io/retrofit/
https://medium.com/@prakash_pun/retrofit-a-simple-android-tutorial-48437e4e5a23
https://en.proft.me/2019/09/28/android-networking-coroutines-and-retrofit/


Thanks.