Android: JetPack, RxJava, and RetroFit

Desenvolvimento de Software e Sistemas Móveis (DSSMV)
Licenciatura em Engenharia de Telecomunicações e Informática
LETI/ISEP

2025/26

Paulo Baltarejo Sousa
pbs@isep.ipp.pt

Instituto Superior de Engenharia do Porto
P.PORTO

Disclaimer

Material and Slides

Some of the material/slides are adapted from various:

- Presentations found on the internet;
- Books;
- Web sites;
- ...

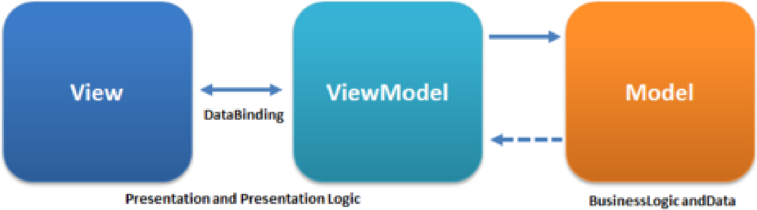
Outline

1. **Architecture Components within Android Jetpack**
   - Room Database
   - Repository
   - LiveData
2. **Introduction To Reactive Programming - RxJava, RxAndroid**
   - Understanding Observables
3. **Retrofit**
   - Retrofit converters and adapters
4. **Bibliography**

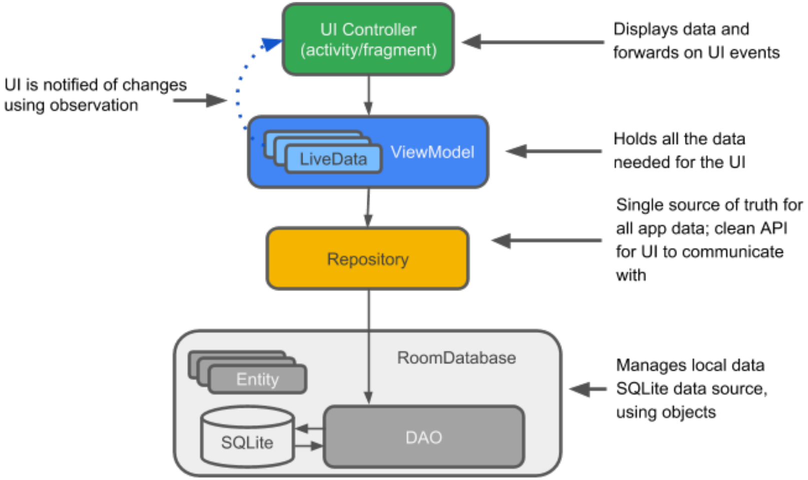
Architecture Components within Android Jetpack

What are Android Architecture Components?

- Architecture Components help you **structure your app** in a way that is robust, testable, and maintainable with less boilerplate code.
- They are part of Android Jetpack
- They promote the Model-View-ViewModel (MVVM) pattern. .



Components (I)



Displays data and forwards on UI events

UI is notified of changes using observation

Holds all the data needed for the UI

Single source of truth for all app data; clean API for UI to communicate with

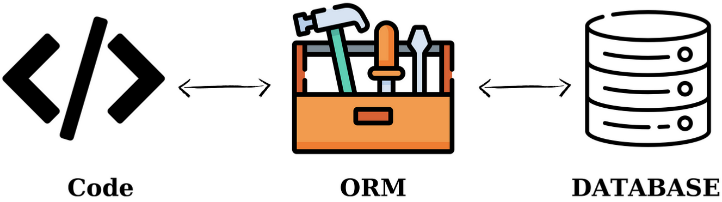
Manages local data SQLite data source, using objects

Components (II)

- **Activity/Fragment**: Activities and Fragments are not part of the "architecture components", but are basic UI components.
- **ViewModel** : Provides data to the UI and acts as a communication layer between the Repository and the UI.
- **LiveData**: is an observable data container which automatically notifies the UI of data changes without requiring an explicit call to the **ViewModel**.
- **Repository**: it handles data operations.
- **Room database**: is a library encapsulating and simplifying access to the SQLite database.
- **Entity**: it is an annotated class that describes a database table.
- **SQLite database**: On the device, data is stored in a SQLite database.
- **DAO** (Data access object). A mapping of SQL queries to functions.

Room Database

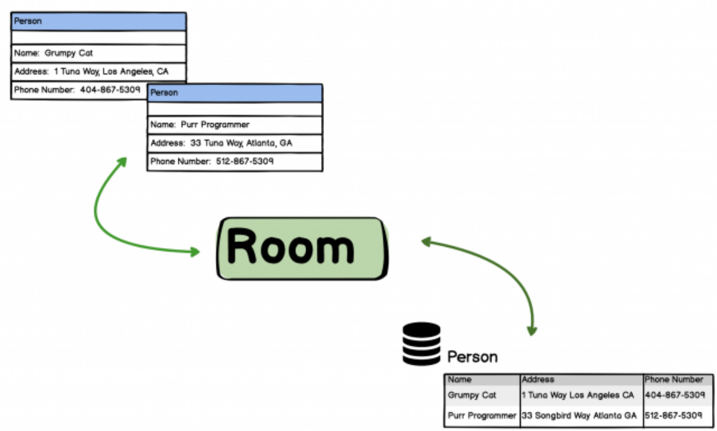
- Room is an **Object Relational Mapping** (ORM) library.



- Room **maps database objects to Java objects**.
- Room has three main components of Room Database :
  - Entity
  - Dao
  - Database

Entity

- Represents a **table within the database**.
    - Room creates **a table for each class that has `@Entity` annotation, the fields in the class correspond to columns in the table**.



Person

Name: Grumpy Cat

Address: 1 Tuna Way, Los Angeles, CA

Phone Number: 404-867-5309

Person

Name: Purr Programmer

Address: 33 Tuna Way, Atlanta, GA

Phone Number: 512-867-5309

Name	Address	Phone Number
Grumpy Cat	1 Tuna Way Los Angeles CA	404-867-5309
Purr Programmer	33 Songbird Way Atlanta GA	512-867-5309

Entities Annotations[1]

- `@Entity`: creates a table
- `@ColumnInfo`: Specify the name of the column in the table.
- `@PrimaryKey`: Every entity needs a primary key.
    - `autoGenerate=true`: auto-generate a unique key for each entity

```
@Entity(tableName = "todo_table")
public class Todo {
  @PrimaryKey(autoGenerate=true)
  private int id;
  @ColumnInfo(name = "user_id")
  private int userId;
  @ColumnInfo(name = "title")
  private String title;
  @ColumnInfo(name = "completed")
  private Boolean completed;
  // Getters and setters are not shown for brevity,
  // but they're required for Room to work if variables are private.
}
```

[1] https://developer.android.com/reference/android/arch/persistence/room/package-summary

DAO

- Each DAO includes methods that offer abstract access to database.
- The DAO must be an `interface` or `abstract` class and is annotated with `@Dao`.
- There are four annotations for methods (to perform CRUD operations): `@Query`, `@Insert`, `@Update`, `@Delete` .

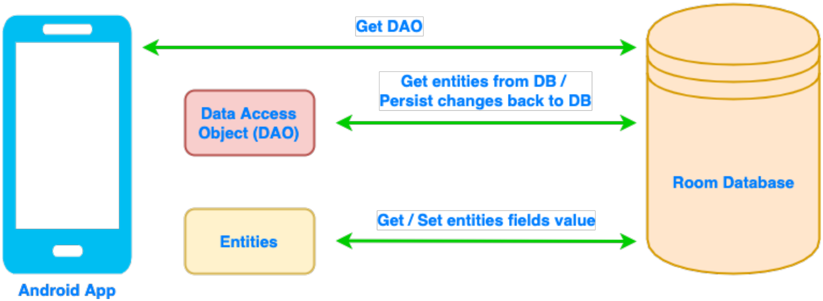
```
@Dao
public interface TodoDao {
  @Query("SELECT * from todo_table")
  LiveData<List<Todo>> getTodoList();
  @Insert
  void insert(Todo todo);
}
```

Database

- To create a database we need to define an `abstract` class that extends `RoomDatabase`.
- This class is annotated with `@Database`, lists the entities contained in the database, and the DAOs which access them.

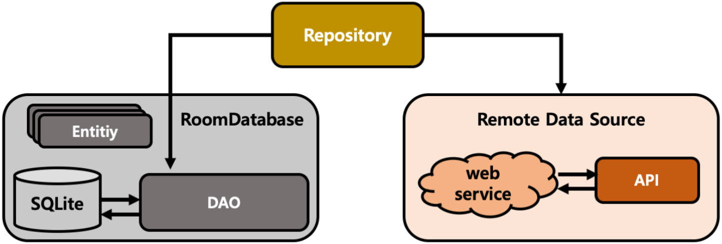
```
@Database(entities = {Todo.class}, version = 1, exportSchema = false)
public abstract class TodoRoomDatabase extends RoomDatabase {
  public abstract TodoDao todoDao();
  private static TodoRoomDatabase INSTANCE = null;
  public static TodoRoomDatabase getInstance(final Context context) {
    if (INSTANCE == null) {
      synchronized (TodoRoomDatabase.class) {
        if (INSTANCE == null) {
          INSTANCE = Room.databaseBuilder(context.getApplicationContext(),
                 TodoRoomDatabase.class, "todo_database").build();
        }
      }
    }
    return INSTANCE;
  }
}
```

Room Database Overview


Get DAO
Data Access Object (DAO)
Get entities from DB / Persist changes back to DB
Entities
Get / Set entities fields value
Android App
Room Database

Repository

- A Repository is a class that **abstracts access to multiple data sources**.



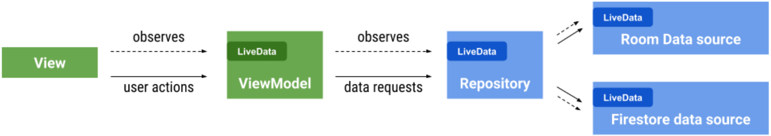
- The Repository is not part of the Architecture Components libraries, but is a suggested best practice for code separation and architecture.
- A Repository class **handles data operations**.

Repository

```
public class TodoRepo {
 private TodoDao todoDao;
 private LiveData<List<Todo>> todoList;
 private static TodoRepo instance;

 public static TodoRepo getInstance(Application application){
  if(instance==null){
   instance = new TodoRepo(application);
  }
  return instance;
 }

 private TodoRepo(Application application) {
  TodoRoomDatabase db = TodoRoomDatabase.getInstance(application);
  todoDao = db.todoDao();
  todoList = todoDao.getTodoList();
 }
 public LiveData<List<Todo>> getTodoList() {
  return todoList;
 }

 ...
}
```

LiveData

- LiveData, which is a lifecycle library class **for data observation**, can help an app respond to data changes.
- LiveData is an observable data holder that is aware of the lifecycle of an Android component.
    - Gets updates to the data **while the Lifecycle is in an active state** (STARTED or RESUMED)
- LiveData is a wrapper that **can be used with any data**, including objects that implement Collections such as List.
    - When you update **the value stored in the LiveData object, it triggers all registered observers**.
        - LiveData allows UI controller observers to subscribe to updates.
        - When the data held by the LiveData object changes, the UI automatically updates in response.



MutableLiveData

- `LiveData` **has no publicly available methods to update the stored data**.
- `LiveData` **is immutable** by default
- `MutableLiveData` is a subclass of `LiveData` that provides mutability, allowing the modification of its value.
  - `MutableLiveData` **is mutable** and is a subclass of `LiveData`.
  - It is commonly used within ViewModels to hold and expose data that can be updated over time.
- The `MutableLiveData` class adds two public methods that allow you to set the value of a `LiveData` object:
  - `setValue(T)`
  - `postValue(T).`

Making data observable with `LiveData` (I)

- To make data observable, wrap it with `LiveData`.
- When you pass data through the layers of your app architecture from a Room database to your UI, that data has to be `LiveData` in all layers.
- All the data that `Room` returns to the `Repository`, and the `Repository` then passes to the `ViewModel`, must be `LiveData`.
- The ViewModel class must have a `LiveData` instance attribute to hold the data.
  - The `ViewModel` is a class whose role is to provide data to the UI and survive configuration changes.
  - It acts as a communication center between the Repository and the UI.
  - A ViewModel holds your app's UI data in a lifecycle-conscious way that survives configuration changes.

Making data observable with `LiveData` (II)

- DAO

```
@Query("SELECT * from todo_table")
LiveData<List<Todo>> getTodoList();
```

- Repository

```
public LiveData<List<Todo>> getTodoList() {
  return ...
}
```

- ViewModel

```
private LiveData<List<Todo>> todoList;
public LiveData<List<Todo>> getTodoList() {
  return todoList;
}
```

Observing LiveData

- Create an observer of the data in `Activity` and override the observer's `onChanged()` method.
- When the `LiveData` changes, the observer is notified and `onChanged()` is executed.

```
LiveData<List<Todo>> liveData = todoViewModel.getTodoList();

liveData.observe(this, new Observer<List<Todo>>() {
  @Override
  public void onChanged(List<Todo> todos) {
    adapter = new TodoListAdapter(MainActivity.this, R.layout.list_item, todos);
    listView.setAdapter(adapter);
  }
});
```

Check: `TP07_01(LocalTodoApp)`

Introduction To Reactive Programming - RxJava, RxAndroid

What is Reactive Programming

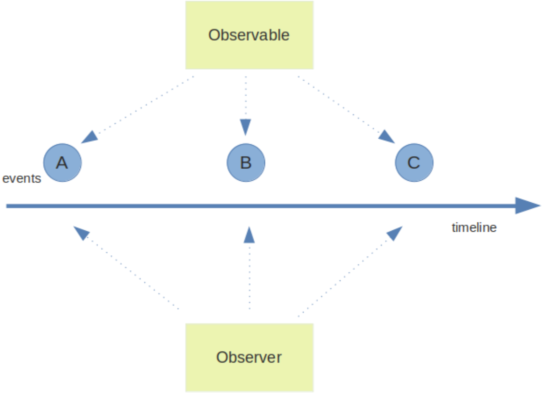
- Reactive Programming is basically **event-based asynchronous programming**.
- Everything **is an asynchronous data stream**, which **can be observed** and **an action will be taken place when it emits values**.
- An advantage of asynchronous approach is, **as every task runs on its own thread**, all the task can start simultaneously and amount of time takes complete all the tasks is equivalent to the longer task in the list.

What is RxJava (I)

- Reactive Extensions (ReactiveX or RX) is a library that **follows Reactive Programming principles**.
- RxJava is Java implementation of Reactive Extension (from Netflix).
    - It is a library **that composes asynchronous events by following Observer Pattern**.
- The core concept of RxJava **is the observable sequence**, which represents a stream of data or events that **can be observed and processed over time**.
    - Observable sequences can be created from a wide range of sources, including asynchronous data sources like network requests and user input, and they can be transformed and manipulated using a variety of operators.
- RxJava also includes a number of other features and tools, such as scheduling, error handling, and utility functions, which make it a powerful and useful library for building reactive applications in Java.

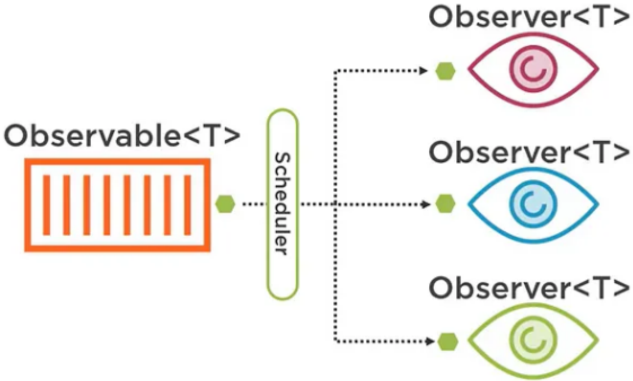
What is RxJava (II)

- An Observable **is a class that represents a stream of data or events** that can be emitted to Observers.
- Observers **can subscribe to Observables to receive the emissions from the Observable**.



What is RxJava (III)

- When events are sent into the Observable, **they must be of or derived from that specific type**.
- Next, each event is sent along to the observers, either synchronously or asynchronously, depending on the requested scheduler



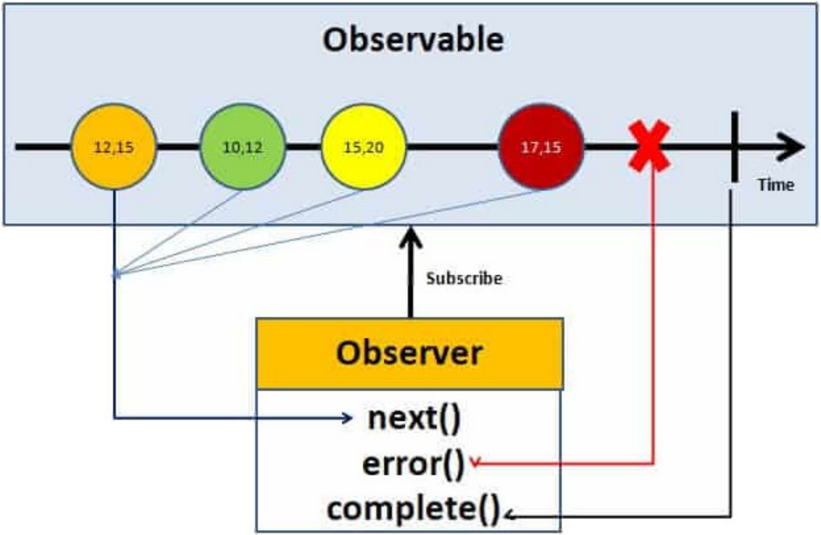
What is RxAndroid

- RxAndroid is specific to Android Platform with few added classes on top of RxJava.
- More specifically, Schedulers are introduced in RxAndroid (`AndroidSchedulers.mainThread()`) which plays major role in supporting multithreading concept in android applications.
- Schedulers basically decides the thread on which a particular code runs whether on background thread or main thread.
- Even through there are lot of Schedulers available,`Schedulers.io()` and `AndroidSchedulers.mainThread()` are extensively used in android programming.

RxJava/RxAndroid: Components (I)

- **Observable**: Observable is a data stream that do some work and emits data.
- **Observer**: Observer is the counter part of Observable. It receives the data emitted by Observable.
- **Subscription**: The bonding between Observable and Observer is called as Subscription. There can be multiple Observers subscribed to a single Observable.
- **Operator / Transformation**: Operators modifies the data emitted by Observable before an observer receives them.
    - The library offers wide range of amazing operators like map, combine, merge, filter and lot more that can be applied onto data stream.
- **Schedulers**: Schedulers decides the thread on which Observable should emit the data and on which Observer should receives the data i.e background thread, main thread etc.,

RxJava/RxAndroid: Components (II)



Adding Dependencies

- To get started, you need to add the RxJava and RxAndroid dependencies to your projects build.gradle and sync the project.

```
val rx_java = "3.1.8"
val rx_android = "3.0.2"

implementation("io.reactivex.rxjava3:rxjava:$rx_java")
implementation("io.reactivex.rxjava3:rxandroid:$rx_android")
```

Steps

1. Create an `Observable` that emits data.

```
Observable<String> animalsObservable = Observable.just("Ant", "Bee", "Cat", "
      Dog", "Fox");
```

2. Create an `Observer` that listen to `Observable`.

```
Observer<String> animalsObserver = new Observer<String>() {
  @Override
  public void onSubscribe(@NonNull Disposable d) {
  }
  @Override
  public void onNext(@NonNull String s) {
  }
  @Override
  public void onError(@NonNull Throwable e) {
  }
  @Override
  public void onComplete() {
  }
};
```

Steps

2. Create an `Observer` that listen to `Observable` (cont...).
    - `onSubscribe()`: Method will be called when an Observer subscribes to Observable.
    - `onNext()`: This method will be called when Observable starts emitting the data.
    - `onError()`: In case of any error, onError() method will be called.
    - `onComplete()`: When an Observable completes the emission of all the items, onComplete() will be called.

Steps

3. Make `Observer` subscribe to `Observable` so that it can start receiving the data

```
animalsObservable.subscribe(animalsObserver);
```

- The data will be emitted and processed by the current scheduler/thread (usually the main thread).

4. Specifying a different thread to execute operations

```
animalsObservable
.subscribeOn(Schedulers.io())
.observeOn(AndroidSchedulers.mainThread())
.subscribe(animalsObserver);
```

- `subscribeOn(Schedulers.io())`: This tell the Observable to run the task on a background thread.
- `observeOn(AndroidSchedulers.mainThread())`: This tells the Observer to receive the data on android UI thread so that you can take any UI related actions.

Check: `TP07_02(localTodoApp2)`

Observables & Observers (I)

- Observable emits data/event and an Observer can receive it by subscribing on to it.
- There are multiple types of Observables, Observers and there are number of ways to create an Observable.
  - Observables:
    - `Observable`
    - `Single`
    - `Maybe`
    - `Completable`
    - `Flowable`
  - Observers:
    - `Observer`
    - `SingleObservable`
    - `MaybeObservable`
    - `CompletableObserver`
- Observables differs from another in the way they produce the data and the number of emissions each Observable makes.

Observables & Observers (II)

- All Observables varies from one another in the number of emission it makes.

Observable	Observer	nr emissions
`Observable`	`Observer`	Multiple or None
`Single`	`SingleObserver`	One
`Maybe`	`MaybeObserver`	One or None
`Flowable`	`Observer`	Multiple or None
`Completable`	`CompletableObserver`	None

Observables & Observers: Use Cases (I)

- `Observable` and `Observer`
  - The Observable that emits more than one value.
    - If the user wants to download a file from the internet, he should be provided with the progress of the upload.
    - The Observable has to emit values at regular intervals.
- `Flowable` and `Observer`
  - Flowable is similar to Observable but this comes into picture when Observable is emitting a huge number of values that cannot be received/consumed by the Observer.
- `Single` and `SingleObserver`
  - Single is used when the Observable has to emit only one value like a response from network call.

Observables & Observers: Use Cases (II)

- `Maybe` and `MaybeObserver`
  - Maybe is used when the observable has to emit a value or no value.
  - It is not recommended much to use Maybe in RxJava for Android Application Development
- `Completable` and `CompletableObserver`
  - Completable is used when the Observable has to do some task without emitting a value

Retrofit

What is Retrofit

- Retrofit [2] is a **REST Client for Java and Android**.
- It makes it relatively easy to retrieve and upload JSON (or other structured data) via a REST based webservice.
- In Retrofit you configure which converter is used for the data serialization.
- Typically for JSON you use GSon, but you can add custom converters to process XML or other protocols.

[2] `https://square.github.io/retrofit/`

How it Works

- Retrofit uses an HTTP client (OkHttp) to execute the network requests.
    - This execution happens on a background thread.
- When OkHttp client receives a response from the server, it passes the response back to Retrofit.
- Retrofit then does its magic: it pushes the meaningless response bytes through converters and wraps it into a usable response with meaningful Java objects.
- This resource-intensive process is still done on a background thread.
- Finally, when everything is ready Retrofit needs to return the result to the UI thread of your Android app.
- By default, this return wrapping is done as `Call<TypedResponseClass>type`.
    - This action of returning from the background thread, which receives and prepares the result, to the Android UI thread is a call adapter!

Using Retrofit

- To work with Retrofit you basically need the following three classes:
    - **Model class** which is used as a JSON model
        - You can generate Java objects based on JSON using the following tool: `http://www.jsonschema2pojo.org/`.
    - Interfaces that define the possible **HTTP operations**
    - **Retrofit.Builder class** - Instance which uses the interface and the Builder API to allow defining the URL end point for the HTTP operations.

HTTP operations (I)

- Every method of an interface represents one possible API call.
- It must have a HTTP annotation (`@GET`, `@POST`, etc.) to specify the request type and the relative URL.
- The return value wraps the response in a `Call` object with the type of the expected result.

```
@GET("users")
Call<List<User>> getUsers();
```

- You can use replacement blocks and query parameters to adjust the URL.
- With the help of the `@Path` annotation on the method parameter, the value of that parameter is bound to the specific replacement block.

```
@GET("users/{name}/commits")
Call<List<Commit>> getCommitsByName(@Path("name") String name);
```

HTTP operations (II)

- Query parameters are added with the `@Query` annotation on a method parameter.

```
@GET("users")
Call<User> getUserById(@Query("id") Integer id);
```

- The `@Body` annotation on a method parameter tells Retrofit to use the object as the request body for the call.

```
@POST("users")
Call<User> postUser(@Body User user)
```

Create the Retrofit Instance

- Create an instance using the `Retrofit.Builder` class and configure it with a base URL.

```
public class RetrofitClientInstance {
 private static Retrofit retrofit;
 private static final String BASE_URL = "https://jsonplaceholder.typicode.com"
       ;

 private static Retrofit getRetrofitInstance() {
  if (retrofit == null) {
   retrofit = new Retrofit.Builder()
   .baseUrl(BASE_URL)
   .addConverterFactory(GsonConverterFactory.create())
   .build();
  }
  return retrofit;
 }

 public static RetrofitApiInterface getRetrofitApiInterface() {
  retrofit = getRetrofitInstance();
  return retrofit.create(RetrofitApiInterface.class);
 }
}
```

Define the Endpoints

- The **endpoints are defined inside of an interface** using special retrofit annotations to encode details about the parameters and request method.

```
public interface GetDataService {
  @GET("/photos")
  Call<List<RetroPhoto>> getAllPhotos();
}
```

Making HTTP Requests (I)

- Instantiate `GetDataService`

```java
RetrofitApiInterface service = RetrofitClientInstance.getRetrofitInstance().
      create(RetrofitApiInterface.class);
Call<List<RetroPhoto>> call = service.getAllPhotos();
call.enqueue(new Callback<List<RetroPhoto>>() {
  @Override
  public void onResponse(Call<List<RetroPhoto>> call, Response<List<RetroPhoto
        >> response) {
    List<RetroPhoto> items = response.body();
    RetroPhotoAdapter adapter = new RetroPhotoAdapter(MainActivity.this, R.
          layout.list_item, items);
    listView.setAdapter(adapter);
  }
  @Override
  public void onFailure(Call<List<RetroPhoto>> call, Throwable t) {
    Toast.makeText(MainActivity.this, "Something went wrong...Please try later!"
          , Toast.LENGTH_SHORT).show();
  }
});
```

Making HTTP Requests (II)

- `enqueue()` asynchronously sends the request and notifies your app with a callback when a response comes back.
  - Since this request is asynchronous, Retrofit handles it on a background thread so that the main UI thread is not blocked or interfered with.
- To use `enqueue()`, you must implement two callback methods:
  - `onResponse()`
  - `onFailure()`

Check: `TP07_03(RemoteTodoApp)`

Retrofit Converters

- Retrofit can be configured to use a specific converter.
- This converter handles the data (de)serialization.
- Several converters are already available for various serialization formats.
  - To convert to and from JSON:
    - Gson: `com.squareup.retrofit:converter-gson`
    - Jackson: `com.squareup.retrofit:converter-jackson`
    - Moshi: `com.squareup.retrofit:converter-moshi`
  - To convert to and from XML:
    - Simple XML: `com.squareup.retrofit:converter-simplexml`

Retrofit Adapters

- Retrofit adapters for modeling network responses
- Retrofit can also be extended by adapters to get involved with other libraries like RxJava 3.x, Java 8 and Guava.

```
Retrofit retrofit = new Retrofit.Builder()
  .baseUrl("https://api.example.com")
  .addCallAdapterFactory(RxJava3CallAdapterFactory.create())
  .build();
```

- With this adapter being applied the Retrofit interfaces are able to return RxJava 3.x types, e.g., Observable, Flowable or Single and so on.

```
@GET("users")
Observable<List<User>> getUsers();
```

Check: `TP07_04(RemoteTodoApp2)`

Adding Dependencies

- To get started, you need to add the retrofit dependency to your projects `build.gradle` and sync the project.

```
val retro = "2.11.0"


implementation("com.squareup.retrofit2:retrofit:$retro")
implementation("com.squareup.retrofit2:converter-gson:$retro")
implementation("com.squareup.retrofit2:adapter-rxjava3:$retro")
```

Bibliography

Resources

- "Mastering Android Application Development", by Antonio Pachon Rui, 2015
- https://developer.android.com/index.html
- http://simple.sourceforge.net/home.php http://simple.sourceforge.net/download/stream/doc/tutorial/tutorial.php