MVVM

What We have learned by previous Video Tutorials :

Data Binding Notification

**MODEL**

**VIEW**

**VIEW MODEL**

**UI LOGIC DATA**

Interaction between View model and View Update in Data or any change notify by base observable

Extends

BaseObservable

Baseadapter validation and other validation we can work

For ListView Model

Model

ListViewModel

ListView or UI

setAdapter

Data Binding BaseObservable for update

InnerLayout

CustomAdapter

@BindingAdapter

Imageview

compile **'com.squareup.picasso:picasso:2.5.2'**

**http://i.imgur.com/DvpvklR.png**

# Data Binding Fragment

Main Activity During the work with fragments we need the binding onCreateView

FRAGMENT2

FRAGMENT3

FRAGMENT1



# MVVM with Remote Server or Local DataBase

ViewLayer ViewModelLayer RequsetManager Layer APIServiceLayer Volley or Retrofit Layer

LOGIN

VIEWMODEL

LOGIN

API SERVICE

LOGIN

API

DATA

MANAGER

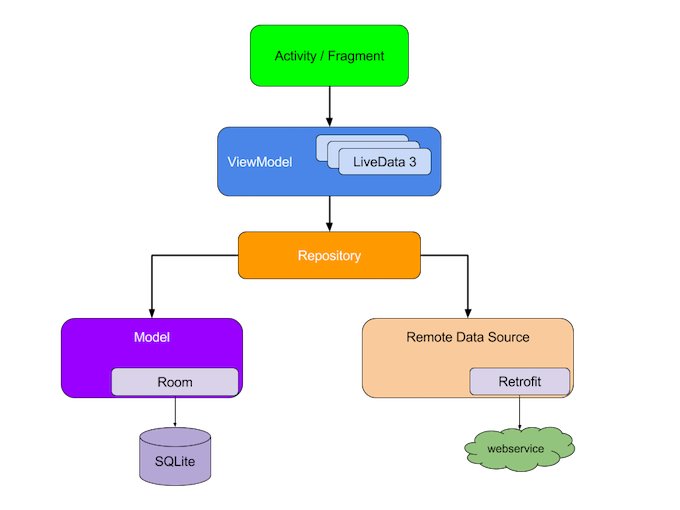
LOGIN

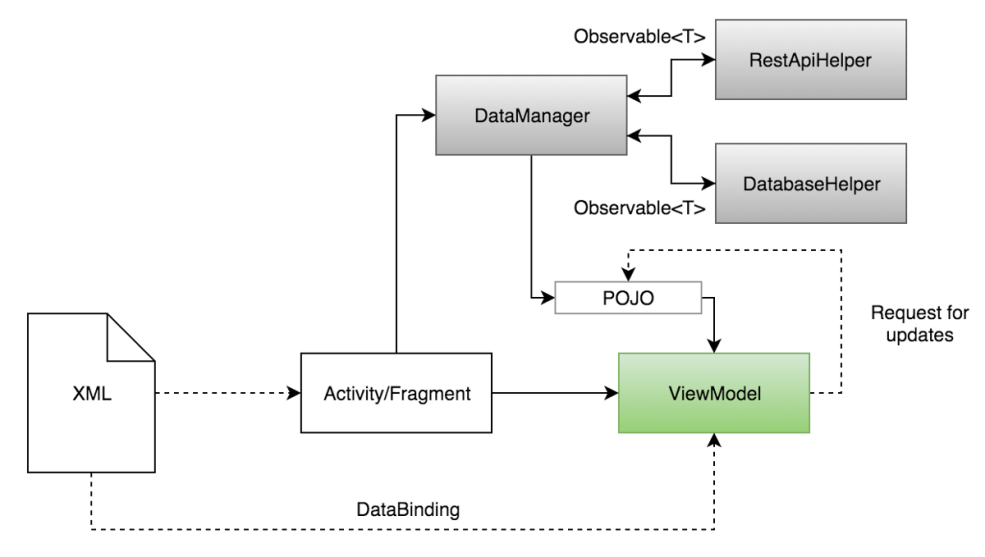
VIEW

DATA BINDING CALL DATA MANAGER CALL SERVICE CALL API Volley or Retrofit

Using Presenter or Call Methods

AUTHENICATE USER





So first I want to show to you in simple steps

**compile** 'com.android.volley:volley:1.0.0'

Today I describe to you how a remote web service works in mvvm as in above design and in coming videos will describe you how we can access using retrofit or rx java and api’s

I design a small example here at a basic level .

VIEW MODEL

XML

LISTVIEW

BUTTON

Data Binding

VOLLEY

MAIN ACTIVITY

DATA MANAGER

INNER

LAYOUT

Volley Request

Call Method

Adapter Volley Response

CUSTOM ADAPTER

Binding using class name

**public static** String *BASEURL* =**"http://pastebin.com/raw/2bW31yqa"**;

{

students:[

{

firstname: "Richard",

lastname: "Levi",

age: "24"

},

{

firstname: "Joana",

lastname: "Daniela",

age: "25"

},

{

firstname: "Paula",

lastname: "Johanna",

age: "22"

},

{

firstname: "Therese",

lastname: "Boje",

age: "23"

},

{

firstname: "Jenny",

lastname: "Jones",

age: "25"

}

]

}

# RETROFIT

**Retrofit** is a REST Client for **Android** and Java by Square. 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.

There is another awesome library [Retrofit](http://square.github.io/retrofit/) to make the http calls. Retrofit is better alternative to volley in terms of ease of use, performance, extensibility and other things.

Retrofit is a REST Client for Android and Java by Square. 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.

To work with Retrofit you need basically three classes.

* Model class which is used to map the JSON data to
* Interfaces which defines the possible HTTP operations
* Retrofit.Builder class - Instance which uses the interface and the Builder API which allows defining the URL end point for the HTTP operation.

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.

@GET("users")

Call<List<User>> getUsers()

 A replacement block is added to the relative URL with {}. 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)

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)

Today Session is based on how simple access of retrofit without databinding , once base clears to you then on next video will show you how retrofit works for Databinding and MVVM

Steps for Retrofit Example

1. Json …
2. compile **'com.google.code.gson:gson:2.6.2'**compile **'com.squareup.retrofit2:retrofit:2.0.2'**compile **'com.squareup.retrofit2:converter-gson:2.0.2' add in gradle**
3. make models using serilaizable as we are using gson so for json serilazation make model fields serialize
4. if you want help for pozo use <http://www.jsonschema2pojo.org/>
5. After making this class put in model ..
6. Create Retrofit Client class using APICALL interface
7. Make custom adapter for inner list
8. Call Retro and get list

CUSTOM ADAPTER

INNER LAYOUT

JSON

API CALL

RETROFIT

CLASS

MAIN ACTIVITY

XML

LISTVIEW

BUTTON

REACTIVE PROGARMING

What is Reactive Programming :

**Languages**

* Java: [RxJava](https://github.com/ReactiveX/RxJava)
* JavaScript: [RxJS](https://github.com/ReactiveX/rxjs)
* C#: [Rx.NET](https://github.com/Reactive-Extensions/Rx.NET)
* C#(Unity): [UniRx](https://github.com/neuecc/UniRx)
* Scala: [RxScala](https://github.com/ReactiveX/RxScala)
* Clojure: [RxClojure](https://github.com/ReactiveX/RxClojure)
* C++: [RxCpp](https://github.com/Reactive-Extensions/RxCpp)
* Lua: [RxLua](https://github.com/bjornbytes/RxLua)
* Ruby: [Rx.rb](https://github.com/Reactive-Extensions/Rx.rb)
* Python: [RxPY](https://github.com/ReactiveX/RxPY)
* Go: [RxGo](https://github.com/ReactiveX/RxGo)
* Groovy: [RxGroovy](https://github.com/ReactiveX/RxGroovy)
* JRuby: [RxJRuby](https://github.com/ReactiveX/RxJRuby)
* Kotlin: [RxKotlin](https://github.com/ReactiveX/RxKotlin)
* Swift: [RxSwift](https://github.com/kzaher/RxSwift)
* PHP: [RxPHP](https://github.com/ReactiveX/RxPHP)
* Elixir: [reaxive](https://github.com/alfert/reaxive)
* Dart: [RxDart](https://github.com/ReactiveX/rxdart)

## ReactiveX for platforms and frameworks

* [RxNetty](https://github.com/ReactiveX/RxNetty)
* [RxAndroid](https://github.com/ReactiveX/RxAndroid)
* [RxCocoa](https://github.com/kzaher/RxSwift)

It is a style of programming where you define a source of data and a consumer of that data. Once you connect the consumer to the source, the library (which in this blog is RxJava) takes care of pushing the data, generated by the source, to the consumer.

The above definition talks about three important things

1. Source of data
2. Consumer of data
3. Connecting Consumer to Source

**Reactive programming is an extension of the**[**Observer software design**](https://en.wikipedia.org/wiki/Observer_pattern)**pattern, where an object has a list of Observers that are dependent on it, and these Observers are notified by the object whenever it’s state changes.**

***Reactive programming****is a*[*programming paradigm*](https://en.wikipedia.org/wiki/Programming_paradigm)*oriented around*[*data flows*](https://en.wikipedia.org/wiki/Dataflow_programming)*and the propagation of change. This means that it should be possible to express static or dynamic data flows with ease in the programming languages used, and that the underlying execution model will automatically propagate changes through the data flow.*

Rx is made up of three key points.

***RX = OBSERVABLE + OBSERVER + SCHEDULERS***

**Observable:** Observable are nothing but the data streams. Observable packs the data that can be passed around from one thread to another thread. They basically emit the data periodically or only once in their life cycle based on their configurations.

**Observers:** Observers consumes the data stream emitted by the observable. Observers subscribe to the observable using subscribeOn()method to receive the data emitted by the observable. Whenever the observable emits the data all the registered observer receives the data in onNext() callback. Here they can perform various operations like parsing the JSON response or updating the UI

**Schedulers:**Remember that Rx is for asynchronous programming and we need a thread management. There is where schedules come into the picture. Schedulers are the component in Rx that tells observable and observers, on which thread they should run.

Real World Example Reactive Programming :

If we go to in ATM (Automated Teller Machine) to withdraw some cash. we insert the card into the machine, enter pin code, enter the amount you want to withdraw and hit Process ..

Means there are two possible outcomes at that time you will get :

Either it will dispense the cash to you. Once all the money is dispensed it will signal to you with a message about the successful transaction completion.

* Or there is no available cash left, in that case it will signal with a message of transaction failure or other failures

So ATM is a source here , of cash, and we are the consumers, of cash. Based on the transaction details the cash flows from the ATM to us on button press..

So the example of reactive programming is or definition is..

## REAL WORLD EXAMPLE

ATM

USER

Transaction

RX JAVA

Observable Subscriber

It is a style of programming where we define a source of data and a consumer of that data. Once we connect the consumer to the source, the library (which in this blog is RxJava) takes care of pushing the data, generated by the source, to the consumer.

**In reactive programming the consumer reacts to the data as it comes in. This is the reason why asynchronous programming is also called reactive programming. Reactive programming allows to propagates event changes to registered observers.**

**So Three tasks are main in Reactive Programming as below :**

* **Source of data**
* **Consumer of data**
* **Connecting Consumer to Source**

## For these task java provide us RxJava

[Reactivex](http://reactivex.io/)  …

[RxJava](https://github.com/ReactiveX/RxJava) is a library that helps programmers to write asynchronous, concurrent, and resilient applications. Using RxJava you write programs in [reactive programming](https://en.wikipedia.org/wiki/Reactive_programming) paradigm..

There are two basic and very important items in reactive programming, [Observables](http://reactivex.io/RxJava/javadoc/rx/Observable.html" \t "_blank)and [Observers](http://reactivex.io/RxJava/javadoc/rx/Observer.html). Observables publish values, while Observers subscribe to Observables, watching them and reacting when an Observable publishes a value.

In simpler terms:

* An Observable performs some action, and publishes the result.
* An Observer waits and watches the Observable, and reacts whenever the Observable publishes results.

### [Observables](http://www.vogella.com/tutorials/RxJava/article.html#observables)

Observables are the sources for the data. Usually they start providing data once a subscriber starts listening. An observable may emit any number of items (including zero items). It can terminate either successfully or with an error. Sources may never terminate, for example, an observable for a button click can potentially produce an infinite stream of events.

### [Subscribers](http://www.vogella.com/tutorials/RxJava/article.html#subscribers)

A observable can have any number of subscribers. If a new item is emitted from the observable, the onNext() method is called on each subscriber. If the observable finishes its data flow successful, the onComplete() method is called on each subscriber. Similar, if the observable finishes its data flow with an error, the onError() method is called on each subscriber.

A class that implements the Observer interface must provide methods for each of the three changes above:

An onNext() method that the Observable calls whenever it wishes to publish a new value

An onError() method that’s called exactly once, when an error occurs on the Observable.

An onCompleted() method that’s called exactly once, when the Observable completes execution.

A class that implements the Observer interface must provide methods for each of the three changes above:

1. An onNext() method that the Observable calls whenever it wishes to publish a new value
2. An onError() method that’s called exactly once, when an error occurs on the Observable.
3. An onCompleted() method that’s called exactly once, when the Observable completes execution.

### [Why doing asynchronous programming](http://www.vogella.com/tutorials/RxJava/article.html#why-doing-asynchronous-programming)

Reactive programming provides a simple way of asynchronous programming. This allows to simplify the asynchronously processing of potential long running operations. It also provides a defined way of handling multiple events, errors and termination of the event stream. Reactive programming provides also a simplified way of running different tasks in different threads. For example, widgets in SWT and Android have to be updated from the UI thread and reactive programming provides ways to run observables and subscribers in different threads.

It is also possible to convert the stream before its received by the observers.

Observable<Integer> source = Observable.range(1, 5);

// defining the consumer

Subscriber<Integer> consumer = new Subscriber<Integer>() {

@Override

public void onNext(Integer number) { System.out.println(number); }

@Override

public void onError(Throwable e) { System.out.println("error"); }

@Override

public void onCompleted() { System.out.println("completed"); }

};

// connecting the consumer to source

source.subscribe(consumer);

Observable<T> represents a source. An Observable can be created using one of the many [factory methods](https://github.com/ReactiveX/RxJava/wiki/Creating-Observables) it provides. Observable.range(int start, int count) is one of them.

Subscriber<T> serves as a consumer of data. RxJava uses onNext(T data) method on the Subscriber to push the data emitted by the source, the Observable, to the consumer of data, the Subscriber.

Once all the data is emitted by the source, RxJava signals the completion using onComplete() method on the Subscriber. In the example above the consumer just prints completely. In the ATM example, completion is signalled using a successful transaction message.

Small example

compile 'io.reactivex:rxandroid:1.2.1'

compile 'io.reactivex:rxjava:1.1.9'

**private void** createObservableAndObserver()  
{  
 **myObservable** = Observable.*create*(**new** Observable.OnSubscribe<String>() {  
 @Override  
 **public void** call(Subscriber<? **super** String> subscriber) {  
 subscriber.onNext(**editText**.getText().toString());  
 subscriber.onCompleted();  
 }  
 });  
  
  
 **myObserver** = **new** Observer<String>() {  
  
 @Override  
 **public void** onCompleted() {  
  
 }  
  
 @Override  
 **public void** onError(Throwable e) {  
  
 }  
  
 @Override  
 **public void** onNext(String text) {  
 **textView**.setText(text);  
 }  
 };  
  
  
}

createObservableAndObserver();  
**button**.setOnClickListener(**new** View.OnClickListener() {  
 @Override  
 **public void** onClick(View view) {  
  
 **myObservable**.subscribe(**myObserver**);  
  
 }  
});

**Observable Types**

|  |  |
| --- | --- |
| Flowable<T> | Emits 0 or n items and terminates with an success or an error event. Supports backpressure, which allows to control how fast a source emits items. |
| Observable<T> | Emits 0 or n items and terminates with an success or an error event. |
| Single<T> | Emits either a single item or an error event. The reactive version of a method call. |
| Maybe<T> | Succeeds with an item, or no item, or errors. The reactive version of an Optional. |
| Completable | Either completes with an success or with an error event. It never emits items. The reactive version of a Runnable. |

Retrofit with Rxjava

compile **'com.google.code.gson:gson:2.6.2'**compile **'com.squareup.retrofit2:retrofit:2.0.2'**compile **'com.squareup.retrofit2:converter-gson:2.0.2'**compile **'io.reactivex.rxjava2:rxjava:2.0.1'**compile **'io.reactivex.rxjava2:rxandroid:2.0.1'**compile **'com.jakewharton.retrofit:retrofit2-rxjava2-adapter:1.0.0'**