# RESEARCH ANDROID

MVP

<https://blog.mindorks.com/android-mvp-architecture-extension-with-interactors-and-repositories-bd4b51972339>

<https://github.com/MindorksOpenSource/android-mvp-architecture>

Mô hình MVP

- Tạo thuận lợi cho unit test

- Tách biệt giữa Model và View trong mô hình MVC

- Tất cả logic dữ liệu đều được đưa vào presenter

MVP chia ứng dụng thành 3 thành phần cơ bản:

- **Model**: xử lý dữ liệu

- **View**: hiển thị dữ liệu và tiếp nhận tương tác người dùng để truyền đến Presenter xử lý

- **Presenter**: cầu nối giữa View và Model.Xử lý tương tác người dùng truyền đến từ View và update Model

Một số quy tắc sử dụng MVP (các nguyên tác có thể sử dụng theo nhiều cách tùy vào developer)

- View vẽ giao diện theo hướng dẫn của Presenter

- View chuyển tương tác người dùng đến Presenter

- View không tương tác trực tiếp với Model

- Presenter chuyển yêu cầu của View đến Model và hướng dẫn View hiển thị với mỗi hành động cụ thế

- Model lấy dữ liệu từ server, database và file system

# Glide/UImageLoader/Picasso/Fresco

Picasso:

<https://viblo.asia/p/su-dung-thu-vien-picasso-de-load-anh-trong-android-eW65Gde9KDO>

<http://square.github.io/picasso/>

Glide

<https://viblo.asia/p/thu-vien-load-anh-glide-cho-ung-dung-android-jaqG0QlevEKw>

<https://viblo.asia/p/gioi-thieu-ve-thu-vien-load-anh-glide-va-so-sanh-voi-thu-vien-picasso-6BkGyKrER5aV>

<https://github.com/bumptech/glide>

UImageLoader

<https://viblo.asia/p/load-anh-trong-android-voi-universal-image-loader-amoG81X1vz8P>

<https://github.com/nostra13/Android-Universal-Image-Loader>

Fresco

<https://viblo.asia/p/android-tim-hieu-ve-thu-vien-xu-ly-va-hien-thi-anh-fresco-DbmvmWjDeAg>

<https://github.com/facebook/fresco>

Đặc điểm

- Thư viện load ảnh

- Đơn giản hóa việc load ảnh mà không cần quan tâm đến decoding, memory, disk caching

- Sử dụng mô hình custom của HttpUrlConnectoin tuy nhiên có thể dễ dàng cho nó hoạt động với Volley hoặc OkHttp

Bảng so sánh

# 

# Retrofit 2

<https://code.tutsplus.com/vi/tutorials/getting-started-with-retrofit-2--cms-27792>

<https://viblo.asia/p/android-library-tim-hieu-retrofit-20-AQrMJVojM40E>

<http://square.github.io/retrofit/>

[Retrofit](https://square.github.io/retrofit/) là một HTTP client type-safe cho Android và Java.

Retrofit giúp dễ dàng kết nối đến một dịch vụ REST trên web bằng cách chyển đổi API thành Java Interface.

Tất cả các yêu cầu GET, POST, PUT, PATCH, và DELETE đều có thể được thực thi.

## Tại sao nên sử dụng Retrofit?

Giảm bớt việc xử lý chức năng như: tạo các kết nối, lưu trữ cache, thử lại cache, phân tích luồng phản hồi, xử lý lỗi,…

## Tạo ra các Model một cách tự động từ dữ liệu json trả về

<http://www.jsonschema2pojo.org/>

<https://chrome.google.com/webstore/detail/postman/fhbjgbiflinjbdggehcddcbncdddomop?hl=en>

# REST

RESTful Web Service là các Web Service được viết dựa trên kiến trúc REST. REST đã được sử dụng rộng rãi thay thế cho các Web Service dựa trên SOAP và WSDL. RESTful Web Service nhẹ, dễ dàng mở rộng và bảo trì.

REST là một bộ quy tắc để tạo ra một ứng dụng Web Service, mà nó tuân thủ 4 nguyên tắc thiết kế cơ bản sau:

- Sử dụng các phương thức HTTP một cách rõ ràng

- Phi trạng thái

- Hiển thị cấu trúc thư mục như các URls

- Truyền tải JavaScript Object Notation (JSON), XML hoặc cả hai.

# RxJava/RxAndroid

<https://viblo.asia/p/cung-hoc-rxjava-phan-1-gioi-thieu-aRBeXWqgGWE>

<https://viblo.asia/p/cung-hoc-rxjava-phan-2-threading-concept-MgNeWWwXeYx>

<https://viblo.asia/p/cung-hoc-rxjava-phan-3-core-operators-mrDkMrpzvzL>

<http://blog.danlew.net/2014/09/15/grokking-rxjava-part-1/>

<http://blog.danlew.net/2014/09/22/grokking-rxjava-part-2/>

<http://blog.danlew.net/2014/09/30/grokking-rxjava-part-3/>

<http://blog.danlew.net/2014/10/08/grokking-rxjava-part-4/>

<https://github.com/ReactiveX/RxAndroid>

<http://reactivex.io/documentation/operators.html>

Đơn giản việc xử lý đa luồng nhưng mặc định nó hoàn toàn không chạy đa luồng

Những thành phần cơ bản nhất là Observable và Subscriber.

- Observable phát ra item.

- Subsriber sẽ sử dụng những item đó.

Observable

- Tạo ra 0 hoặc nhiều item

- Gọi đến onNext tương ứng với số item nó có

- Gọi đến onCompleted khi hoàn thành hoặc onError khi xảy ra lỗi tùy thuộc vào kết quả của việc kết thúc

Subsriber

- Nhận item từ Observable

- Gọi onNext khi Observable phát item

- Gọi đến onCompleted khi hoàn thành hoặc onError khi xảy ra lỗi tùy thuộc vào kết quả của việc kết thúc

Concept threading mặc định là:

- Các phần tính toán để tạo nên Observable gốc được chạy trên thread mà hàm subscribe() được gọi đến.

- Các phần tính toán của 1 Operator được chạy trên thread mà Operator ở trước nó được chạy. Nếu trước nó không có 1 Operator nào khác, nó sẽ được chạy trên thread tạo ra Observable gốc.

- Các phần tính toán của 1 Subscriber được chạy trên thread mà Operator ở trước nó được chạy. Nếu trước nó không có 1 Operator nào khác, nó sẽ được chạy trên thread tạo ra Observable gốc.

Rx cung cấp cơ chế xử lý đa luồng Scheduling

- immediate(): Tạo ra và trả về 1 Scheduler để thực thi công việc trên thread hiện tại.

- trampoline(): Tạo ra và trả về 1 Scheduler để sắp xếp 1 hàng chờ cho công việc trên thread hiện tại để thực thi khi công việc hiện tại kết thúc.

- newThread(): Tạo ra và trả về 1 Scheduler để tạo ra 1 thread mới cho mỗi đơn vị công việc.

- computation(): Tạo ra và trả về 1 Scheduler với mục đích xử lý các công việc tính toán, được hỗ trợ bởi 1 thread pool giới hạn với size bằng với số CPU hiện có.

- io(): Tạo ra và trả về 1 Scheduler với mục đích xử lý các công việc không mang nặng tính chất tính toán, được hỗ trợ bởi 1 thread pool không giới hạn có thể mở rộng khi cần. Có thể được dùng để thực thi các tiến trình bất đồng bộ không gây ảnh hưởng lớn tới CPU.

Hàm để dùng các Scheduling trên

- subscribeOn(): Nhận vào tham số là 1 Scheduler, sẽ quyết định việc xử lý các phần tính toán để tạo nên 1 Observable trên thread cung cấp bởi Scheduler đó. Vị trí gọi subscribe() không quan trọng. Nếu gọi nhiều subscribeOn() thì cũng chỉ có hàm gọi đầu tiên có tác dụng. Hàm này sẽ có ảnh hưởng đến việc tạo Observable gốc nên cần chọn đúng dựa theo use case. Trong hầu hết các trường hợp như network request hay các công việc liên quan đến database thì chúng ta sử dụng Schedulers.io()

- observeOn(): Nhận vào tham số là 1 Scheduler sẽ làm cho các Operator hay Subscriber được gọi đằng sau nó chạy trên thread được cung cấp bởi Scheduler đó. Nếu gọi nhiều observeOn() thì các Scheduler có tác dụng đến khi gặp observeOn() tiếp theo. Một trong những điểm mạnh của hàm này là nó cho phép chúng ta thực thi tác vụ trên nhiều thread khác nhau. Tuy vậy thì đối với use case phổ biến nhất trong lập trình Android, đó là update kết quả lên UI ở trong hàm subscribe() thì cần phải sử dụng RxAndroid và đảm bảo là hàm observeOn(AndroidSchedulers.mainThread()) là hàm gọi cuối cùng trước khi gọi đến subscribe()

Không nhầm lẫn giữa subscribeOn() và observeOn(): subscribeOn() có tác dụng với Observable. observeOn() có tác dụng với các Operator và Subscriber được gọi sau nó.

RxAndroid là phiên bản mở rộng của RxJava. RxAndroid cũng cấp cho RxJava thêm 1 Scheduler là AndroidSchedulers. Với hàm mainThread() từ Scheduler này, chúng ta có thể thực thi các tác vụ liên quan đến UI vì nó sẽ đảm bảo là code được execute trên main thread của Android.

**Operators By Category**

Creating Observables

- Form: Nhận para không hạn chế có thể thuộc mọi type. Gọi onNext bằng

số para hoặc kích thước mảng. Và chuyển đổi thành Observable.

- Just: Tương tự form nhưng khi truyền vào mảng thì sẽ phát ra mảng luôn

chứ không phát ra từng phần tử trong mảng

- Defer: Tạo mới Observable cho mỗi Subscriber mà nó có để dữ liệu luôn mới

- Interval: Observable sẽ phát ra 1 item kiểu Long sau mỗi khoảng delay

- Create: Tạo Observable thủ công. Luư giá trị của item khi được subscribe chứ không phải khởi tạo.

- Empty: Tạo Obserable không phát item nhưng chấm dứt bình thường

- Never: Tạo Obserable không phát item cũng không chấm dứt

- Throw: Taoj obserable: Không phát item và kết thức với lỗi

- Range: Tạo Observable phát ra chuỗi các số nguyên cách nhau bởi một khoảng thời gian nhất định

- Repeat: Tạo Observable phát ra một item cụ thể nhiều lần

- Start: Tạo Observable phát item trả về giá trị chỉ thị chức năng

- Timer: Tạo Observable phát ra một item cụ thể sau khoảng thời gian nhất định

Transforming Observables

- Bufer: Thu thập item phát ra từ Observable thành gói và phát ra gói này chứ không phải từng item

- FlatMap: Biến đổi dữ liệu phát ra từ Observable thành Observables sau đó trả về Observable khác

- GroupBy: Chia Observable thành tập hợp Observables mà mỗi phần tử phát ra một tập con items khác từ Observable ban đầu

- Map: Biến đổi dữ liệu phát ra từ Observable thành type T

- Scan: Áp dụng một function cho mỗi item phát ra bởi observable, tuần tự và phát ra mỗi item giá trị tiếp theo ????????????????????????????????????????????

- Window: Định kỳ phân chia các items từ một Observable thành Observable windows và phát ra windows chứ không phải item ??????????????????????????????????

Một số Tranforming Operator thông dụng

- .filter: lọc item phát ra bởi 1 Observable dựa trên điều kiện mà được định nghĩa

- .toList: khi filter xong sẽ có rất nhiều item riêng lẻ khi đó toList dùng để gom tất cả vào trong 1 list

- .take: phát ra số mục chỉ định. nếu nhận được ít hơn thì sẽ kết thúc sớm.

# GreenDAO3

<https://mindorks.com/blog/powerful-android-orm-greendao-3-tutorial>

<https://github.com/greenrobot/greenDAO>

<http://greenrobot.org/greendao/>

Thư viện giúp tương tác với SQL một cách đơn giản như việc thực thi với các đối tượng

Dùng để lưu trữ, cập nhập, xóa, và truy vấn SQL dựa trên các đối tượng Java, sử dụng một định hướng đối tượng API đơn giản

Generate file DaoMaster và DaoSession bằng cách thêm @Entity trước class rồi chọn Build->Make project

Khi thêm bảng thì cần up schemaVersion để override lại hàm onUpgrade trong DaoMaster.OpenHelper cập nhật lại databese nếu không sẽ bị lỗi chưa tạo bảng.

Relations: <http://greenrobot.org/greendao/documentation/relations/#Bi-Directional_1N_Relations>

# EventBUS

<https://viblo.asia/p/giao-tiep-ben-trong-mot-ung-dung-android-voi-eventbus-MdZGAQxjGox>

<https://github.com/greenrobot/EventBus>

<http://greenrobot.org/eventbus/>

Giúp điều phối các sự kiện tránh hệ thống quá tải khi sử lý nhiều sự kiện và quản lý chúng.

Tối ưu hóa và đơn giản hóa các event cũng như trao đổi dữ liệu giữa các Activity, Fragment, Sevice, Threads,... với nhau

Quản lý các sự kiện hay điều khiển các quá trình xử lý dữ liệu, giúp bắt và lấy sự kiện của quá trình đó tại bất kỳ đâu Activity, Fragments

Post Event: nếu không có subriber được tìm thấy, event này sẽ được bỏ đi.

Post Sticky Event: event được lưu trong bộ nhớ đệm. Khi một Activity mới hoặc một Fragment mới theo dõi sự kiện, EventBus sẽ lấy ra sticky event gần đây nhất trong bộ nhớ đệm thay vì thực hiện lại đăng kí một event mới, cho nên event này vẫn nằm trong bộ nhớ đệm sau khi subcriber đã nhận được nó.

EventBus có 4 kiểu gửi event:

- PostThread: đây là kiểu mặc định, đơn giản khi gửi sự kiện. Và nó không cần thiết chạy trên một Main Thread.

- MainThread: sự kiện sẽ được gọi từ Android’s main thread. Và sẽ được gọi một cách trực tiếp

- BackgroundThread sư kiện sẽ được gọi trong background thread. và nó sẽ được gọi trực tiếp ở nơi tạo sự kiện. và EventBus chỉ sử dụng 1 background thread để xử lý

- Async: sự kiện này luôn độc lập với main thread. Vì thế tránh sử dụng nhiều cùng 1 lúc.EventBus cũng sử dụng 1 thread pool để có thể tái sử dụng lại

# Dependency Inversion, Inversion of Control and Dependency Injection

<https://viblo.asia/p/dependency-inversion-inversion-of-control-and-dependency-injection-qzakzNYBkyO>

Dependency Inversion: Các moudle, class cấp cao không nên phụ thuộc vào mudule, class cấp thấp hơn trực tiếp mà nên giao tiếp với nhau thông qua một abstractoin (interface). Bởi lẽ đó, Inversion of Control ra đời để làm nhiệm vụ đó.

Inversion of Control (IoC)

- Là design pattern để hiện thực nguyên lý thuyết kế Dependency Inversion.

- Không quan tâm đến Service được thực hiện như thé nào mà chỉ quan tâm đến những gì mà nó cũng cấp thông qua abstration

- Nơi lưu trữ các implementation của các abstraction muốn truyền vào high-level module

- Có nhiều cách implement IoC như Service Locator, Event, Dependency Ịnection

Dependency injection (DI)

- Là design patern để thực hiện Dependency Inversion, là một trong những subtype của IoC

- Nguyễn tắc cơ bản của DI là làm cho high-level module phụ thuộc vào low-level thông qua injector

- Ưu điểm

+ Giảm sự phụ thuộc giữa các module

+ Code dễ bảo trì dễ thay thế

+ Rất dễ test và viết unit test

+ Dễ dàng thấy quan hệ giữa các module

- Khuyết điểm:

+ Khó debug, do không biết chính xác module nào được gọi

+ Các đối tượng được khởi tạo ngay từ đầu, làm giảm hiệu suất

+ Làm tăng độ phức tạp của code cho những project nhỏ

# Dagger 2

<https://adityaladwa.wordpress.com/2016/05/11/dagger-2-and-mvp-architecture/>

<https://viblo.asia/p/dependency-injection-voi-dagger-2-trong-android-Az45bAYwlxY>

<https://viblo.asia/p/introduction-dependency-injection-on-android-with-dagger-2-n6BkGynXv5aV>

<https://viblo.asia/p/dependency-injection-in-android-with-dagger-2-NbmebapKGYO>

<https://github.com/google/dagger>

Dagger 2 API (một số annotation đặc biệt)

- @Scope: annotation định nghĩa vòng đời tồn tại của Object graph, nó thật sự hữu ích trong việc quản lý vòng đời của các Service sẽ được cung cấp cho Client. Chúng ta có thể define các loại Scope khác nhau như một annotation mới và sử dụng trong từng trường hợp cụ thể

- [@Module](https://viblo.asia/u/Module): cho các lớp mà có phương thức cung cấp sự phụ thuộc.

- [@Providers](https://viblo.asia/u/Providers): cho các phương thức bên trong các lớp [@Module](https://viblo.asia/u/Module).

- @Singleton: được tạo ra 1 lần và đươc sử dụng trong toàn ứng dụng

- [@Inject](https://viblo.asia/u/Inject): để yêu cầu một sự phụ thuộc ( một hàm khởi tạo, một trường, hoặc một phương thức).

- [@Component](https://viblo.asia/u/Component): là cầu nối interface giữa các modules và injection.

- @Qualifier: được cung cấp bởi javax inject package, và được dùng để xác định rõ dependency. Ví dụ, một class có thể yêu cầu cả application context và activity context, nhưng cả 2 object này đều trả về kiểu Context, do đó Dagger2 sẽ cấu hình việc biến nào sẽ provide cho cái nào. @Qualifier được dùng nhiều để phân biệt giữa các object cùng loại khác nhau (như activity context và application context)

- @Name: tương tự @Qualifier. Ngoài ra cung cấp String để đặt tên cho các class object cùng loại.

Dagger 2 workflow

- Xác định các đối tượng phụ thuộc và các phụ thuộc của nó.

- Tạo một class với annotation [@Module](https://viblo.asia/u/Module), sử dụng annotation [@Provider](https://viblo.asia/u/Provider) cho mọi phương thức mà trả về một sự phụ thuộc.

- Yêu cầu phụ thuộc vào các đối tượng phụ thuộc bằng cách sử dụng annotation [@Inject](https://viblo.asia/u/Inject).

- Tạo một interface bằng cách sử dụng annotation [@Component](https://viblo.asia/u/Component) và thêm các class với annotation [@Module](https://viblo.asia/u/Module) được tạo ra trong bước 2.

- Tạo một object của interface [@Component](https://viblo.asia/u/Component) để khởi tạo đối tượng phụ thuộc với các phụ thuộc của nó.

Kotlin

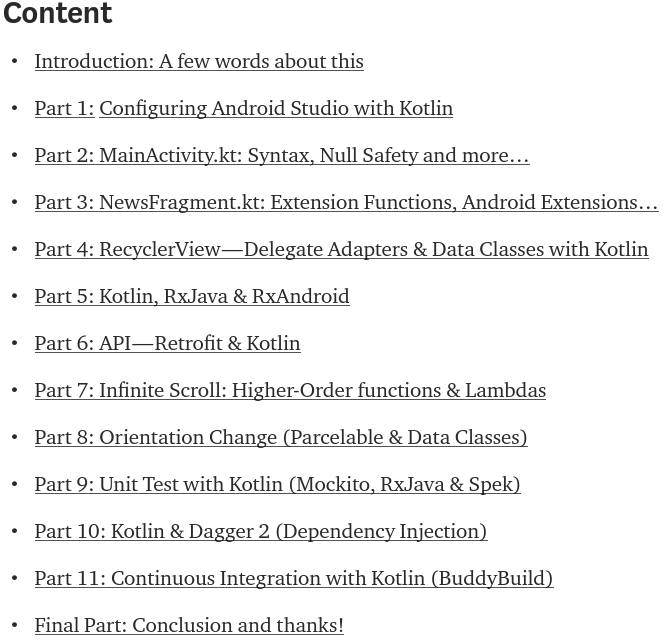
<http://kotlinlang.org>

<https://try.kotlinlang.org>

<https://fabiomsr.github.io/from-java-to-kotlin/>

<http://www.jsoneditoronline.org/>

<https://android.jlelse.eu/learn-kotlin-while-developing-an-android-app-introduction-567e21ff9664>



Ngắn gọn

Thay thế Extend and Implement bằng ":"

Định nghĩa functions bằng "fun" và sử dụng "Unit" nếu không muốn trả về giá trị

Dùng var để khai báo biến. val khai báo hằng. loại biến sẽ được suy ra từ nội dung. cũng có thể chỉ định rõ ràng trong lúc khai báo

Mọi thứ trong kotlin là đối tượng nên không sử dụng kiểu đối tượng cơ bản như double mà dùng Double. Để thực hiện trình biên dịch sẽ biến đổi kiểu đối tượng này thành cơ bản

Sử dụng Fields trong code nhưng nó không được gọi trực tiếp mà thông qua method

Mọi thứ trong Kotlin không phải nullble trừ khi được chỉ định ("?" có thể null hoặc không)

Gọi an toàn: dấu "?" sẽ đánh dấu cho phép lấy giá trị trong trường hợp nó tồn tại



Kiểm tra null trước khi thực hiện các câu lệnh tiếp theo context?.let {...}

Elvis Operator "?:" Thay thế giá trị khi khi đối tượng null

Android Extensions: apply plugin: 'kotlin-android-extensions' thay thế findViewById()

Data Classes: data class RedditNewsItem(var author: String, var title: String)

Có thể sử dụng biến trong chuỗi "$years years"

Để tạo một đối tượng không cần dùng từ khóa "new"

Kotlin cho phép tạo ra một loạt các con số (Int, Long và Char types) bằng cách sử dụng một biểu thức như sau: "1..10"

Lazy tránh khởi tạo thuộc tính nullable

Một hàm bậc cao là một hàm có chức năng như các tham số, hoặc trả về một hàm.

'lateinit' cho phép định nghĩa một thuộc tính không phải là nullable và thiết lập giá trị sau đó.

Mặc định class trong kotlin đều là final

Hàm được đánh dấu override thì nó là open

Trong interface các member mặc định là open

có 4 loại Visivility Modifiers trong Kotlin: private, protected, internal và public

Nếu không chỉ rõ thì mặc định là public

Các chỉ định truy cập: pricate, protected và public giống cách sử dụng trong java

Riêng internal ta có thể sử dụng trong cùng module

Kotlin cho phép thay đổi vị trí tham số trong hàm

**Data Classes with Parcelable**

#### **-** Companion object: Trong Kotlin không co đối từ khóa "static" để tạo thành viên cho lớp nhưng lại tồn tại companion object cho phép cung cấp hành vi tương tự

- @JvmField Annotation: Các lớp implementing Parcelable interface cũng phải có một trường static non-null được gọi là CREATOR của một kiểu implements Parcelable.Creator interface. Để làm cho việc CREATOR implementation của chúng ta có thể nhìn thấy như một trường trong Java, chúng ta cần annotation đặc biệt này được gọi là "@JvmField", nếu không nó sẽ không được tìm thấy và ở giữa quá trình nó sẽ ném ra một ngoại lệ.

#### - Extension Function for the Creator

+ Inline Functions: Chức năng chính nó và chức năng thông qua nó, sẽ được inlined vào các call site, có nghĩa là implement manuall chức năng này tại mọi nơi đã sử dụng nó. Trình biên dịch sẽ tạo ra bytecode cụ thể ở mọi nơi sử dụng nó (giống như "sao chép và dán" chức năng này ở tất cả các nơi đang sử dụng nó) cho hiệu suất tốt hơn và tránh tạo các lớp mới giống như một chức năng mở rộng bình thường có thể làm.

+ Reified T: Truy cập vào một type như là một tham số. Với reified modifier bây giờ nó có thể truy cập bên trong các chức năng, gần như thể nó là một lớp bình thường và không có sự phản chiếu cần thiết!

+ Crossinline: Nếu cần phải thực thi chức năng như là tham số trong một context khác thì crossinline cho phép sử dụng nó theo cách này và cũng có thể chạy nó trong một tiến trình khác. Trong trường hợp này, là bên trong đối tượng Creator.

#### - Secondary Constructors: Constructors phụ kế thừa từ primary constructors

Auto Parcelable: <https://github.com/nekocode/android-parcelable-intellij-plugin-kotlin>

Improve Code Inspection with Annotations

<https://developer.android.com/studio/write/annotations.html#res-annotations>

Nullnes annotations

- @Nullable: Cho phép null

- @NonNull: Không được phép null

Resource annotations

- [@DrawableRes](https://developer.android.com/reference/android/support/annotation/DrawableRes.html), [@DimenRes](https://developer.android.com/reference/android/support/annotation/DimenRes.html), [@ColorRes](https://developer.android.com/reference/android/support/annotation/ColorRes.html), and [@InterpolatorRes](https://developer.android.com/reference/android/support/annotation/InterpolatorRes.html) được thêm bằng định dạng annotation giống nhau và chạy trong quá trình kiểm tra mã

- @AnyRes tham số annotation có thể là bất ký loại R resource

- @ColorRes xác định rằng một tham số nên là một color resource, nhưng một color integer (RRGGBB or AARRGGBB format) không được công nhận là một color resource.

- @ColorInt xác định một tham số là phải là một color integer

Thread annotations

- Các annotations được hỗ trợ: @MainThread, @UiThread, @WorkerThread, @binderThread, @AnyThread

- Nên sử dụng annotation @UiThread với các method được liên kết với multiple views on different threads. Annotation @MainThread với các method được liên kết với app's lifecycle

Value constraint annotations

- @IntRange, @FloatRange, @Size: Xác định hợp lệ của giá trị

- @IntRange: Xác định giá trị tham số integer or long trong giới hạn chỉ định

- @FloatRange: Xác định giá trị tham số float or double trong giới hạn chỉ định

- @Size: Kiểm tra kích thước của collection or array, cũng như độ dài của một string

Permission annotations (@RequiresPermission): Xác định tính hợp lệ khi gọi một method.

Indrect permissions

- Khi permisstion phụ thuộc vào giá trị cụ thể được cung cấp từ một method's parameter, sử dụng @RequiresPermission với paramater đó nhưng không liệt kê permissions rõ ràng.

- Khi sử dụng indrect permissions, các công cụ build phân tích luồng dữ liệu để kiểm tra xem các đối số được truyền vào phương thức này có bất kỳ @RequiresPermission annotions. Sau đó sẽ bất buộc sự tồn tại của annotations từ paramater của method đó.

Return value annotatoins (@CheckResult): Xác định tính hợp lệ của kết quả một method hay giá trị trả về thực sự được sử dụng

CallSuper annotations (@CallSupper): Xác định tính hợp lệ của overriding method gọi super implementation của method.

Typedef annotations

- @IntDef và @StringDef: Tạo các annotations liệt kê của integer và string để xác thực các kiểu mã tham chiếu

- Typedef annotations đảm bảm cho một paramater cụ thể, giá trị trả về, hoặc trường tham khảo một bộ hằng số cụ thể.

- Cho phép hoàn thành mã để tự động cung cấp các hằng số được phép

- Typedef annotations sử dụng @interface để khai báo annotation type

Enable combining constans with flags: Nếu users có kết hợp các hằng được phép với một flag như (|, &, ^, ...), có thể định nghĩa một annotation với một flag để kiểm tra xem một paramater or giá trị trả về tham chiều một pattern có hợp lệ hay không.

Keep annotations (@Keep)

- Đảm bảo một annotation class or method không bị xóa khi code là minified vào build time

- Thường được thêm vào các methods và classes được truy cập thông qua reflection để ngăn không cho trình biên dịch xử lý code như không sử dụng

Code visibility annotatios: Sử dụng các annotations bên dưới để biểu thị khả năng hiển thị của các phần cụ thể của code, như các methods, classes, fields, or packages

- Make visible for testing (@VisibleForTesting): Chỉ ra annotated method rõ ràng là cần thiết hơn bình thường để làm cho method kiểm chứng được. Annotation này có một đối số tùy chọn khác cho phép chỉ định khả năng hiển thị của method đó là gì.

- Restrict an API (@RestrictTo): Chỉ ra quyền truy cập API bị hạn chế

+ Subclasses (@RestrictTo(RestrictTo.Scope.SUBCLASSES)): Hạn chế quyền truy cập API cho các lớp con. Chỉ có classes extend annotated có thể truy cập API.

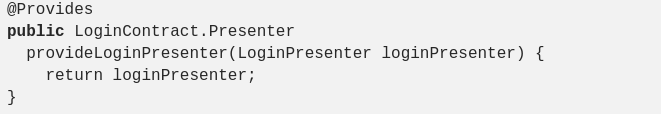
+ Libraries (@RestrictTo(RestrictTo.Scope.GROUP\_ID)): Hạn chế quyền truy cập vào libraries của bạn. Chỉ code library của bạn mới được phép truy cập vào annotated APi. Điều này cho phép bạn không chỉ tổ chức mã của bạn trong bất kỳ package nào bạn muốn mà còn chia sẻ code giữa một nhóm thư viện liên quan.

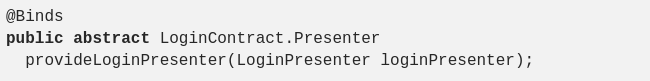
+ Testing (@RestrictTo(RestrictTo.Scope.TESTS)): Ngăn các developers khác truy cập vào testing APIs của bạn. Chỉ code testing của bạn mới được truy cập vào annotated API. Điều này ngăn các developers khác sử dụng API cho phát triển mà bạn chỉ muốn dùng cho thử nghiệm.

Dagger 2 Annotations: @Binds& @ContributesAndroidInjector<https://proandroiddev.com/dagger-2-annotations-binds-contributesandroidinjector-a09e6a57758f>

Binds (@Binds)

- Thay thế các @Provides methods mà chỉ trả lại injected parameter. Sử dụng @Binds annotation và làm method abstract. Chúng ta cũng cần đánh dấu Module là abstract trong trường hợp này sẽ hiệu quá hơn là một Module bình thường.





- Lý do làm abstract Module hiệu quả hơn

+ @Provide methods là instance methods và cần một instance của module để gọi

+ Nếu Module là abstract và chứa các @Binds methods, dagger sẽ không mô tả lại module và thay vào đó trực tiếp Provide của injected parameter

- Module chứa cả @Provide và @Binds thì sẽ có 2 lựa chọn

+ Đánh dấu @Provide instance methods as static

+ Nếu cần giữ instance methods, thì có thể chia module thành hai phần và giải nén tất cả các phương thức @Binds vào một abstract Module

@ContributesAndroidInjector ???

- Generates an AndroidInjector for the return type of this method

- This annotation must be applied to an abstract method in a Module that returns a concrete Android framework type

- Bản mẫu được tạo ra khi sử dụng @ContributesAndroidInjector



+ Tạo LoginActivitySubcomponent

+ Thêm @Subcomponent annotation cần thiết

+ Thêm một entry (LoginActivity.class, LoginActivitySubcomponent.Builder) vào Map của Injector Factories được sử dụng bởi DispatchingAndroidInjector. Dagger-Android uses entry này để build LoginActivitySubcomponent và thực hiện injections vào LoginActivity

+ Đồng thời, binds LoginActivity với object-graph

- Scoping with @ContributesAndroidInjector

Secure data in Android — Encrypting Large Data

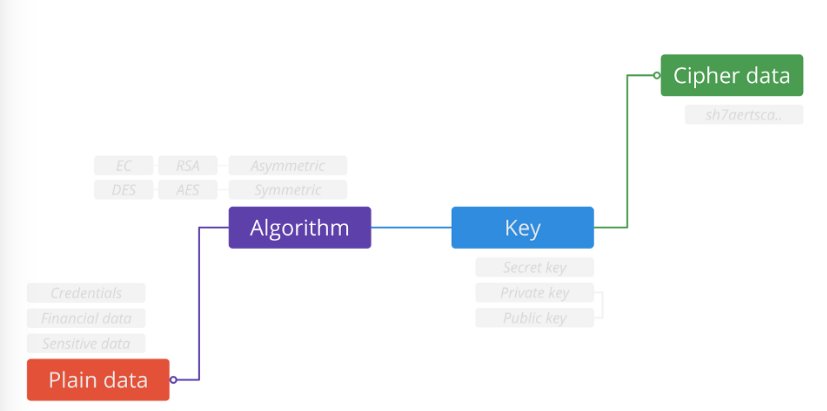
<https://proandroiddev.com/secure-data-in-android-encrypting-large-data-dda256a55b36>



Encryption

- Để đọc encrypted dataần key hoặc pass để decrypt nó

- Unencrypted data gọi là plain data, encrypted data gọi là cipher data



Algorithm Types

- Symmetric

+ Kỹ thuật lâu đời và đơn giản nhất.

+ Enctyption key và decryption key là như nhau

- Asymmetric

+ Các thuật toán sử dụng một cặp khóa (công khai và riêng tư) và sử dụng một thành phần khác của cặp khóa cho các bước khác nhau của thuật toán

+ Phổ biến nhất là RSA - một thuật toán mã hóa công cộng và tiêu chuẩn để mã hóa dữ liệu qua internet

- Stream cipher: Một symmetric encryption algorithm xử lý data một bit hoặc một byte trong một time với một key dẫn đến một cipher data hoặc plain data ngẫu nhiên

- Block cipher:

+ Thuật toán xác định hoạt động trên các nhóm bit chiều dài cố định.

+ Là những thành phần quan trọng trong thiết kế giao thức bảo mật và được sử dụng rộng rãi để mã hóa dữ liệu số lượng lớn

- Modes & Paddings: Block cipher có Modes và padding khác nhau làm tăng mức độ bảo mật

+ Modes: Một mode hoạt động mô tả làm thế nào để áp dụng hoạt động cipher's single-block để an toàn chuyển đổi số lượng dữ liệu lớn hơn một block.

+ padding: Block cipher hoạt động trên những đơn vị có kích thước cố định (được gọi là block size), nhưng thông điệp có chiều dài không xác định. Vì vậy một số chế độ (cụ thể ECB và CBC) yêu cầu final block được đệm trước khi encryption

~ ECB: Electtronic Codebook, đơn giản nhất trong các encryption modes. Thông điệp được chia thành các block và mỗi block encrypted riêng

~ CBC: Cipher Block Chaining, mỗi cipher data block phụ thuộc vào tất cả plain data block xử lý đến thời điểm đó. Để làm cho mỗi thông điệp duy nhất, một vector khởi tạo phải được sử dụng trong first block.

+ Thuật toán not symmetric không có nghĩa nó không có modes và paddings. Ví dụ thuật toán RSA có thể sử dụng với ECB mode và PKCS1Padding

- Key Types

+ Secret key: Một single secret key được sử dụng trong mã hóa đối xứng thông thường để encrypt và decrypt một thông điệp

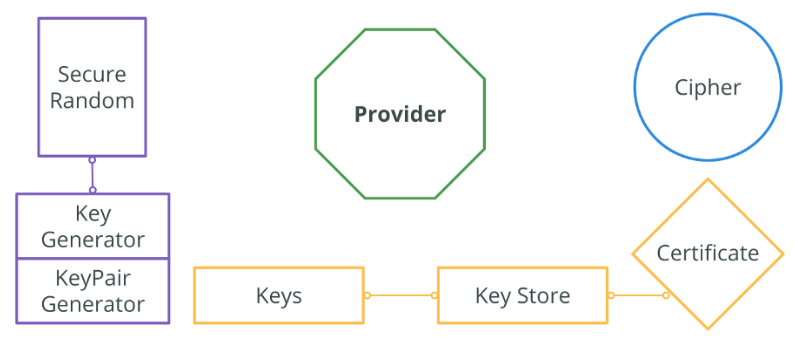
+ Private key: Secret component của một cặp cryptographic keys sử dụng để decryption trong asymmetric cryptography

+ public key: Public component của một cặp cryptographic keys sử dụng để decryption trong asymmetric cryptography

+ Public key và Private key kết hợp với nhau để tạo thành cặp public-private cryptographic key

- Java Cryptography Architecture (JCA)

+ Android build trên JCA cung cấp API cho chữ ký số, chứng chỉ, mã hóa, tạo phím và quản lý



+ KeyGenerator: Cung cấp puclic APi để tạo symmetric cyptographic keys

+ KeyPairGenerator: Một engine class có khả năng tạo ra một private key và public key liên quan sử dụng thuật toán đã được khởi tạo

+ SecretKey: Môt secret (symmetric) key. Mục đích của interface này là nhóm (và cũng cấp an toàn cho) tất cả secret key interfaces (eg: SecretKeySpec)

+ PrivateKey: Một private (asymmetric) key. Mục đích của interface này là nhóm (và cũng cấp an toàn cho) tất cả private key interfaces (eg: RSAPrivateKey)

+ KeyPair: Lớp này đơn giản giữ một cặp khóa (một public key và một private key). Nó không bắt buộc bất kỳ bảo mật nào và khi khởi tạo nên xử lý như một Private key

+ SecureRandom: Tạo ra mật mã an toàn từ những con số ngẫu nhiên. Được sử dụng rộng rãi bên trong KeyGenerator, KeyPairGenerator components và Keys implementations

+ KeyStore: Database với cơ chế bảo vệ dữ liệu tốt, được sử dụng để lưu, nhận và xóa keys. Yêu cầu nhập mật khẩu và có mật khẩu cho mỗi keys. Nó là tập tin được bảo vệ cần tạo, đọc và cập nhật (với API được cũng cấp)

+ Certificate: certificate sử dụng để xác nhận và lưu asymmetric keys

+ Cipher: Cung cấp truy cập tới việc thực hiện của cryptographic ciphers để encryption, decryption, wrapping, unwrapping và signing

+ Provider: Định nghĩa tập hợp các triển khai mở rộng các API độc lập. Provides là nhóm các thuật toán khác nhau hoặc tùy chỉnh. Có provides bên thứ 3, như là [Bouncy Castle](https://www.bouncycastle.org/) and [Spongy Castle](https://rtyley.github.io/spongycastle/) (android version of Bouncy Castle), cũng như provides có sẵn, như là phiên bản rút gọn của Bouncy Castle

- Android Key Store

+ Hệ thống Android Key Store cho phép lưu trữ cryptographic keys trong một container để khó lấy ra khỏi thiết bị

+ Một khi các keys trong key store, chúng có thể được sử dụng cho các crytographic operations với các tài liệu chính còn lại non-exportable

+ Hơn nữa, nó cung cấp thiết bị để tiếp kiệm thời gian và cách sử dụng keys, chẳng hạn như yêu cầu user authentication để sử dụng key hoặc hạn chế các keys chỉ được sử dụng trong các chế độ crytographic

+ AndroidKeyStore thực hiện một JCA Provider, ở:

~ Không có mã KeyStore cần thiết

~ Key material không bao giờ tham gia quá trình ứng dụng

~ Key material có thể ràng buộc secure hardware

~ Asymmetric keys có sẵn từ 18+

~ Symmetric keys có sẵn từ 23+

+ Không có AndroidKeyStore, trước Android 18, bạn sẽ cần tạo một file ở đâu đó trong local của bạn hoặc external device storage, ở đó tất cả keys có thể giữ (dễ dàng extract file). Nhưng với AndroidKeyStore bạn không cần tạo bất cứ thứ gì, hệ thống sẽ quản lý tất cả.

~ Nếu divice manufacture hỗ trọ thực thi moi trường (TTE), keys sẽ được lưu ở đó (lựu chọn an toàn nhất)

~ Nếu divice manufacture không hỗ trợ TTE, keys sẽ được lưu trong môi trường phần mềm mô phỏng, cung cấp bởi hệ thống.

+ Trong cả hai trường hợp, keys sẽ tự động xóa khỏi hệ thống sau khi xóa ứng dụng. Keys material không bao giờ được exposed, ngay cả với bạn.

Unit Test

<https://viblo.asia/p/nhung-dieu-luu-y-khi-viet-unit-test-the-magic-tricks-of-testing-aRBvXxqbvWE>

<https://riggaroo.co.za/introduction-automated-android-testing/>

<https://riggaroo.co.za/retrofit-2-mocking-http-responses/>

<https://medium.com/@fabioCollini/android-testing-using-dagger-2-mockito-and-a-custom-junit-rule-c8487ed01b56>

Mục đích của việc viết init test

- Thorough: Mỗi unit test sẽ chỉ tập trung vào một đối tượng (single object) của app và đảm bảo đối tượng đó hoạt động đúng như mong đợi.

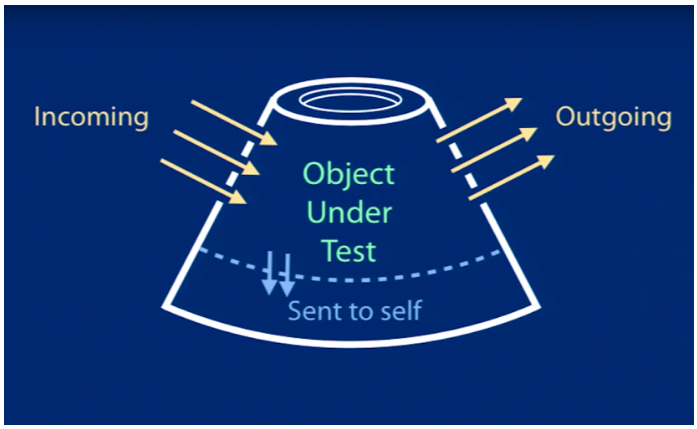
- Stable: Unit Test sẽ hoạt động ổn định, không bị hỏng mỗi khi thay đổi code implement chi tiết.

- Fast: Unit Test cần phải nhanh để có thể kiểm tra các thay đổi nhanh chóng

- Few: Code cho Unit Test cần ở mức vừa đủ, không thừa để tránh chi phí bảo trì (và đọc nữa).

Test gì trong unit test

- Mô tả 3 luồng message



+ Incoming: Các message mà đối tượng có thể nhận được từ các đối tượng khác (từ bên ngoài)

+ Outgoing: Các message mà đối tượng có thể gửi đến các đối tượng khác (ra bên ngoài)

+ Sen to Seft: Các message mà đối tượng dùng để gửi trong chính mình

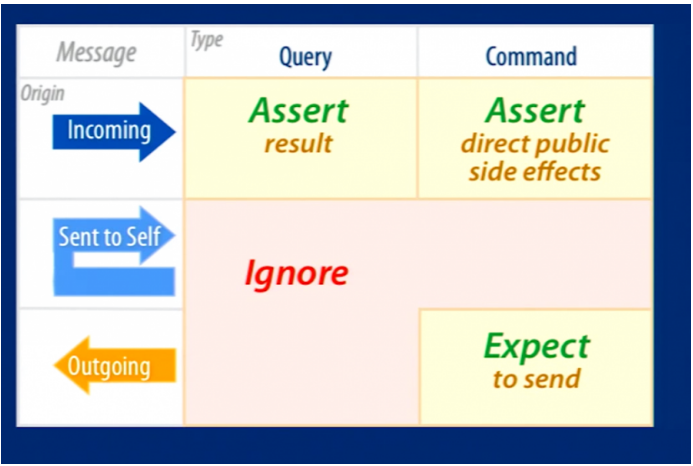
- Một message có 2 loại:

+ Query: Trả về một cái gì đó mà không thay đổi bất cứ thứ gì, không tạo ra side-effect (vd: hàm cộng 2 số nguyên, trả về kết quả của phép tính mà không thay đổi bất kỳ giá trị nào của đối tượng)

+ Command: Không trả về bất cứ thứ gì và thay đổi một vài thứ, có side-effect (vd: khi gọi hàm lưu vào cơ sở dữ liệu, đối tượng có sự thay đổi và các đối tượng khác cũng như app có thể thấy sự thay đổi đó)

- Có một loại message nữa bao gồm cả hai loại Query và Command, ví dụ là hàm pop ddược gọi từ một queue, lúc này sẽ có một giá trị trả về là một đối tượng trong query (query type), đồng thời query đó cũng sẽ được thay đổi, bớt đi một giá trị (command type)

- Các viết unit test cho 2 loại message là khác nhau nên cần phân biệt để viết cho phù hợp. Nếu message gồm cả 2 loại thì viết cả 2 unit test.



Tại sao nên viết unit test

- Kiểm tra khả năng tư duy theo hướng khác thường và làm sạch code trong tiến trình

- Tự tin hơn về code nếu gặp phải các bài kiểm tra

- Biết được trạng thái code của mình được hiển thị chi tiết ở Shiny green status bar and cool reports

- Regression testing được thực hiện dễ dàng hơn, vì automated tests sẽ nhận lỗi trước

Tools được sử dụng

- [JUnit](http://junit.org/): Kiểm tra bình thường

- [Mockito](http://mockito.org/): mocking out các lớp khác không được test

- PowerMock – mocking out các lớp tĩnh như Android Evironment class

Vị trí của automated tests trong android app

- androidTest: Android Instrumentation Tests ở đây. Cần chạy trên Android Emulator hoặc physical device

- test/: Unit Tests ở đây. Chaỵ trên JVM local và không chạy trên android device hoặc emulator. Điều này có là nó không có quyền truy cập vào Android classes

Annotation

- Phương thức gắn annotation @Before sẽ chạy trước khi chạy các phương thức gắn annotaion @Test. Phương thức này dùng để khai báo định nghĩa dữ liệu cần thiết trước khi tiến hành kiểm thử

- Các phương thức gắn annotation @Test được viết để chạy unit test

DEMO

Link Demo: <https://github.com/HoangKimTuan/ResearchAndroidVNG>

Xây dựng mô hình bằng Dagger và MVP

Sử dụng Retrofit để lấy dữ liệu từ <http://jsonplaceholder.typicode.com>

Sử dụng Glide để load hình ảnh về ListView

Sử dụng EventBUS để chuyển dữ liệu giữa 2 activity

Sử dụng GreenDAO để lưu các comment và photo offline

Sử dụng RxJava để xử lý dữ liệu được lấy bởi Retrofit

Demo RxJava: Project RxJava

Demo Kotlin: Project KotlinOnAndroid

Demo Unit Test: Project RobolectricUnitTest, UnitTestApiRequest, Dagger2MockitoJUnitRuleAndroidTest