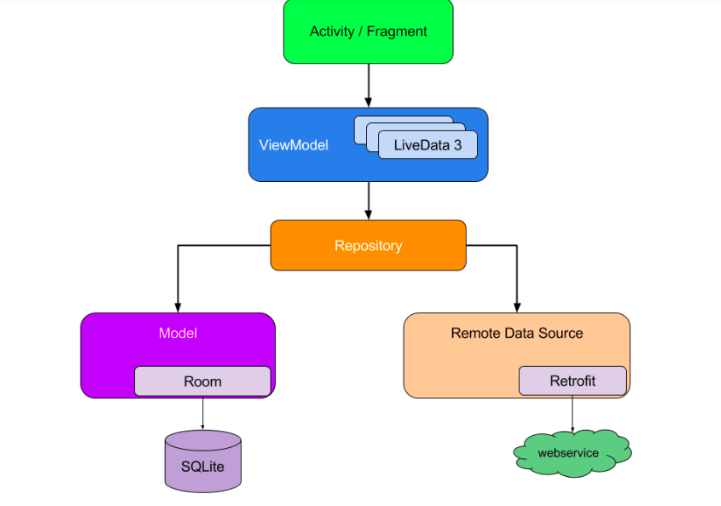
**BÁO CÁO THỰC TẬP**

|  |  |
| --- | --- |
| **Mentor** | : Lê Trí |
| **Sinh viên** | : Hà Văn Dương |
| **Mã sinh viên** | : B15DCCN160 |
| **Doanh nghiệp thực tập** | : VNPT-IT |

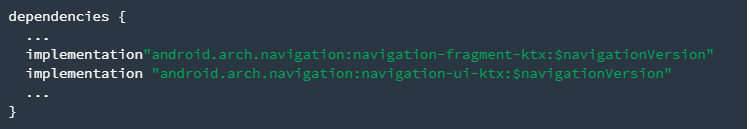
**Architecture Component**



1. Navigation

* Trong ứng dụng android có nhiều fragment và để giải quyết vấn đề chuyển qua lại giữa các fragment ta sử dụng Navigation.

1. Add Navigation
   1. Add dependency navigation



* Add Navigation Graph vào trong project: Create file navigation.xml trong folder navigation. Navigation Graph là một biều đồ để mô tả các Navigation Destination và kết nối chúng với nhau. Navigation Destination tương ứng với các screen.
  1. Create NavigationHostFragment
* Add <fragment> trong layout\_activity. NavigationHostFragment điều hướng các fragment trong ứng dụng. Khi di chuyển giữa các **destination** định nghĩa bên trong NavigationGraph thì NavigationHostFragment sẽ hoán đổi các các fragments ra vào khi cần thiết.
  1. Add Fragments(Destination) vào trong NavigationGraph
* Add các fragments sau đó tạo action giữa các fragments. Xủ lý sự kiện click để chuyển từ fragment này sang fragment khác.
  1. Change Back button
* Thông thường khi ta nhấn back thì sẽ trở lại màn hình trước đó. Trong một số trường hợp ta không muốn trở lại màn hình trước thì ta sẽ modify

+ Các hành vi sẽ được lưu lại trong **back stack** của hệ thống. Muốn remove destination nào ra khỏi back stack thì sử dụng popToUp ở action của destination đó.

* 1. Add Navigation Drawer
* Có 2 kiểu để navigation drawer xuất hiện:

+ Vuốt từ bên trái màn hình ra

+ Click vào drawer icon trên actionbar

* Navigation Drawer là một phần của MaterialComponent library nên khi sử dụng thì ta implement nó trong dependencies

1. Safe Args Plugin

* Truyền dữ liệu giữa các Fragment ta sử dụng Bundle. Bundle dạng key-value. Fragment A lưu dữ liệu dạng key-value xong gửi sang Fragment B. Fragment B sẽ có key để so khớp. Nhưng kĩ thuật này có thể có lỗi như không tìm được key. Để bắt được lỗi ta sử dụng **Safe Args** trong Navigation Architecture Conponent.

1. ViewModel
2. UIController

* Một UIController là một lớp dựa trên giao diện người dùng như là Activity hoặc Fragment. Một UIController nên chỉ chứa các xử lý logic tương tác với UI và các tương tác của hệ điều hành như là display views và thu thập dữ liệu người dùng.

1. ViewModel

* Một ViewModel sẽ *lưu dữ liệu được hiển thị* trong activity và fragment mà liên kết với ViewModel. Một ViewModel có thể thực hiện các *phép tính và biến đổi đơn giản trên dữ liệu* để hiển thị bởi UI Controller (update data lên UI thì đặt trong UI Controller). Mỗi ViewModel sẽ liên kết với một fragments

1. ViewModelFactory.

* Khi các ViewModel có param thì sử dụng ViewModelFactory, sẽ khởi tạo ViewModel object.

Thực hiện:

B1: Add ViewModel class

* + Add dependency:



* + Create kotlin class ViewModel

B2: Override onCleared()

* + ViewModel sẽ bị hủy khi liên kết với fragment bị tách ra, hoặc *khi activity đã kết thú*c. Trước khi ViewModel bị hủy thì onCleared() callback sẽ được gọi để làm sạch tài nguyên.

B3: Liên kết ViewModel với Fragment

* + Một ViewModel cần liên kết với một UIController. Để tạo liên kết, ở UI Controller tạo tham chiếu đến ViewModel.

B4: Khởi tạo ViewModel

* + Trong quá trình thay đổi cấu hình như *rotate device*, thì UI Controller như là fragment đã khởi tạo lại. Tuy nhiên *object ViewModel* vẫn tồn tại. Vì vậy ViewModel là một nơi rất tốt để lưu dữ liệu khi thay đổi cấu hình.
* Tóm tại: Tạo object ViewModel từ ViewModelProvider.

Vậy ViewModelProvider hoạt động như thế nào:

+ ViewModelProvider sẽ trả về một ViewModel hiện có nếu nó tồn tại, hoặc nó sẽ tạo ra cái mới nếu chưa tồn tại

+ ViewModelProvider tạo ra thực thể ViewModel liên kết với một Activity hoặc một Fragment.

1. LiveData
2. LiveData

* **Mục đích:** Sử dụng LiveData để cập nhật sự thay đổi của Data lên UI theo thời gian thực

**Observer Patter:** sẽ có một đối tượng trung tâm gọi là subject, nó nắm danh sách các đối tượng quan sát khác (đối tượng quan sát chính là các observer). Để khi nào subject có bất kì thay đổi nào về trạng thái thì sẽ thông báo đến các observer.

**Live Data** cũng tương tự như vậy cũng dựa trên mô hình Observer. Subject ở đây là thuộc tính trong ViewModel (LiveData object).

* + Sử dụng LiveData để cập nhật dữ liệu theo thời gian thực cho UI

+ ViewModel sử dụng LiveData để bao các thuộc tính. Các thuộc tính sẽ không còn kiểu **Int** nữa mà nó sẽ sử dụng kiểu **MutableLiveData**. **MutableLiveData** cũng chính là LiveData nhưng nó cho *phép cập nhật sửa chữa các giá trị*.

1. DataBinding with LiveData and ViewModel
   * Tích hợp data binding với LiveData trong ViewModel objects. Tự động giao tiếp giữa các view trong layout và object ViewModel, và cho phép đơn giản code sử dụng LiveData
   * Layout sẽ giao tiếp trực tiếp với dữ liệu bên trong ViewModel object mà không thông qua Fragment UI. Ta sẽ truyền ViewModel object vào trong data binding, thì sẽ tự động hóa giao tiếp giữa các views và ViewModel object
   1. Add ViewModel vào data binding

B1: Thêm ViewModel vào data binding

* Liên kết ViewModel với cả layout tương ứng. Ở layout tương ứng với ViewModel, trong <data> khai báo thêm <variable>

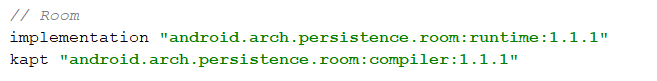
B2: Sử dụng listener bindings để xử lý sự kiện

* Listener bindings : là các biểu thức ràng buộc khi chạy các sự kiện như onClick(), onZoomIn(), onZoomOut() được kích hoạt.
* Data binding tạo ra 1 listener và đặt listener trong view.
  1. Add LiveData cho Data binding.
* Add các LiveData vào trong các views để hiển thị lên UI.
  1. LiveData Tranformations
* Khi truyền data muốn map (ánh xạ) hoặc tranform(chuyển đổi) dữ liệu. Tranform các LiveData bằng cách sử dụng các method trong **Tranformations** Class là:

+ Tranformations.map

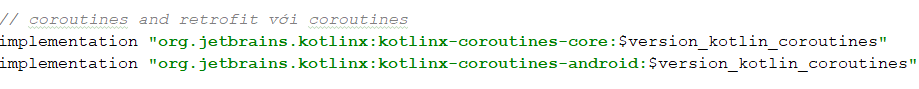
+ Tranformations.switchMap

1. Room
2. Create Room
   * Implement Room



* + Room là một thư viện cơ sở dữ liệu, giúp cho ứng dụng tương tác với cơ sở dữ liệu.Là một phần của Android Architecture Components
  1. ***Create Entity***
  + Dữ liệu ở ViewModel sẽ được lưu vào trong Database (1 Room Database)
  + **Entity**: Một entity đại diện cho một object và các thuộc tính của nó được lưu trữ trong database. ***Một entity class định nghĩa ra một bảng***, và mỗi thực thể trong đó đại diện cho 1 hàng trong bảng. Mỗi thuộc tính là một cột.
  + **Query**: request để lấy dữ liệu hoặc thông tin từ 1 bảng hoặc kết hợp các bảng, hoặc thực hiện một hành động nào đó với dữ liệu như: update,delete
  1. ***Create a DAO (Interface)***
  + Cung cấp các method insert, delete, update database tương tác với database
  + Khi call method trong DAO, thì Room sẽ sinh ra tất cả code cần thiết để thực thi với database
  1. ***Create Room Database***
  + Create một abstract class extends RoomDatabase . Add annotation @Database với các arguments.
  + Định nghĩa DAO bên trong Database
  + Annotaion @Volatile thể hiện biến giá trị của biến sẽ không lưu trong cached, mà tất cả đọc và ghi sẽ lưu trong main memory.
  + Create một singleton database object

1. Corountines and Room
   * Implement Corountines



* + Sử dụng các coroutines cho các task cần nhiều thời gian như các hoạt động với database.
  + Coroutines là bất đồng bộ và non-blocking

+ Coroutines chạy độc lập với luồng thực thi chính của chương trình.

* + Trong Coroutine có một thành phần là:

+ **Job**: Sử dụng Job để hủy bỏ coroutine.

+ **Dispatcher**: sẽ gửi các Coroutine chạy trên các thread khác nhau. Ví dụ: Dispatcher.Main sẽ chạy trên Main thread. Dispatcher.IO sẽ chạy với các hoạt động I/O.

+ **Scope**: Định nghĩa một context, tức là xác định Coroutines sẽ chạy trên thread nào

**Pattern**:

* + Launch một coroutine chạy trên main thread hoặc UI thread.
  + Gọi một **suspend function** để không phải dừng UI thread trong khi chờ kết quả và cho chạy trên I/O thread để thực hiện thao tác với database.

1. RecycleView
2. RecycleView
   * RecyclerView sẽ hiển thị một list hoặc một grid data. Lợi ích của RecyclerView là rất tốt cho các list lớn. Khi muốn hiển thị dạng gridlayout ở views thì thêm số cột.
   * Create class ViewHolder để hiển thị dữ liệu đã load rồi khi scrolllist bằng cách extends lại viewHolder trong RecyclerView và tạo một layout\_custom. Tức là với những view đã khởi tạo rồi khi scroll sẽ bị mất, khi scroll về thì sẽ được gọi lại chứ không phải tạo lại.
   * Create Adapter đưa data hiển thị lên RecyclerView. Adapter sẽ implements 3 method:

+ getItemCount() sẽ trả về số lượng item trong list

+ onCreateViewHolder() sẽ trả về ViewHolder, khởi tạo

+ onBindViewHolder(): gán dữ liệu và các views

1. DiffUtil and DataBinding with RecycleView

**2.1 DiffUtil**

- Sử dụng DiffUtil để hiển thị list lớn và dễ mở rộng, sửa đổi.

* + Việc sử dụng notifyDataSetChanged() không hiệu quả trong việc hiển thị, nó sẽ thực hiện với toàn bộ danh sách. Đối với những list phức tạp thì gây ra đơ, trải nghiệm không tốt khi người dùng scoll.

🡪 Để khắc phục điều này thì ta sẽ chỉ cập nhật những view đã thay đổi, thì cần phải biết chính khác những gì đã thay đổi. Sử dụng DiffUtil để tính toán ra được sự khác biệt giữa 2 list. Cần implement 2 method:

+ areItemsTheSame() return true nếu là các item giống nhau và ngược lại

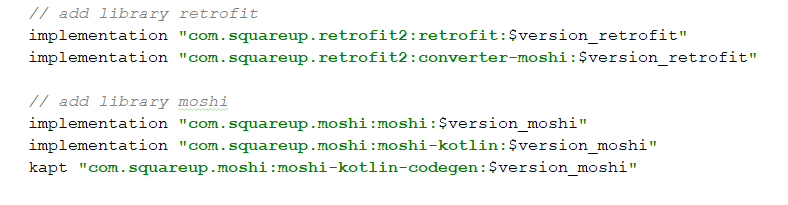
+ areContentsTheSame() so sánh giữa 2 list về data

* 1. **Sử dụng ListAdapter** để quản lý list.
  + ListAdapter sẽ theo dõi list và thông báo adapter khi list updated. Sẽ thay thế cho RecyclerView.Adapter. Tạo Adapter kế thừa từ ListAdapter

+ Tạo class DiffCallBack implements từ **DiffUtil** để thực hiện các method so sánh chênh lệch giữa list cũ và list mới để biết được những item nào đã được thêm, sửa, xóa.

+ ListAdapter cung cấp method gọi là submitList() để updated item thay đổi

1. Retrofit and Moshi
   * Sử dụng **Retrofit library** để kết nối tới một **REST web service** trên internet và lấy một response
   * Sử dụng **Moshi** library để parse từ **JSON response** sang **data object** .
   * Implement Retrofit và Moshi



1. Connect web service sử dụng Retrofit
   * Retrofit cần 2 điều có sẵn để build một web service API. API này sẽ dùng để kết nối tới service kia thông qua method GET lấy dữ liệu. Đó là:
     + base URL của server
     + converter factory 🡪 Nó sẽ nói cho Retrofit sẽ làm gì với data lấy từ web service. Retrofit sẽ lấy một JSON response từ web service và return ra một String. Ta có ScalarsConverter sẽ hỗ trợ convert kiểu dữ liệu nguyên thủy và string.
2. Parse JSON response sử dụng Moshi
   * ConverterFactory trong Retrofit làm việc được với Moshi
   * Sử dụng annotation **@Json** để phân biệt giữa property của class và key name trong JSON
3. Sử dụng Coroutines với Retrofit
   * Sử dụng coroutines để code hiệu quả và dễ đọc hơn.

+ Ta thêm addCallAdapterFactory() thì return của method GET không phải là Call mà có thể thay thế object khác. Khi sử dụng Coroutines ta sẽ thay là Deferred object.

1. Display an image internet
   * Add library **Glide** : để load và display image từ Url.
   * Tạo binding adapter và call Glide

+ binding adapter sẽ định nghĩa việc chuyển đổi string url sang uri object rồi sang ImageView.

* + Add **loading** and **error** với images: bằng cách thêm apply() vào trong Glide sau method load(). Trong apply() nếu đang loading image thì hiển thị icon loading, còn đang bị lỗi thì hiển thị icon error

1. Repository
   * Khi chạy ứng dụng load dữ liệu từ internet xuống thì sẽ bị chậm vì vậy ta cần sử dụng offline caching để load dữ liệu nhanh tăng trải nghiệm người dùng.
   * ***Offline caching*** là ứng dụng lưu data từ internet vào bộ nhớ cục bộ của thiết bị để truy cập nhanh hơn và có thể truy cập dữ liệu khi không có internet. Room là nơi lưu cache data
   * Dữ liệu từ trên network sau đó đưa vào offline cache rồi hiển thị lên trên screen. Như vậy giảm được thời gian load. 🡪 trải nghiệm người dùng tốt. Khi offline thì vẫn có thể load dữ liệu từ
   * Sử dụng repository để quản lý offline cache. Repository sẽ có những xử lý logic để lấy kết quả từ network và sau đó cập nhật vào database, và getAllData từ database ra rồi convert sang domain model để hiển thị.