Faster development and testing using Kotlin, MVVM and Dagger

Waheed Nazir

mwaheednazir8@gmail.com
https://github.com/WaheedNazir

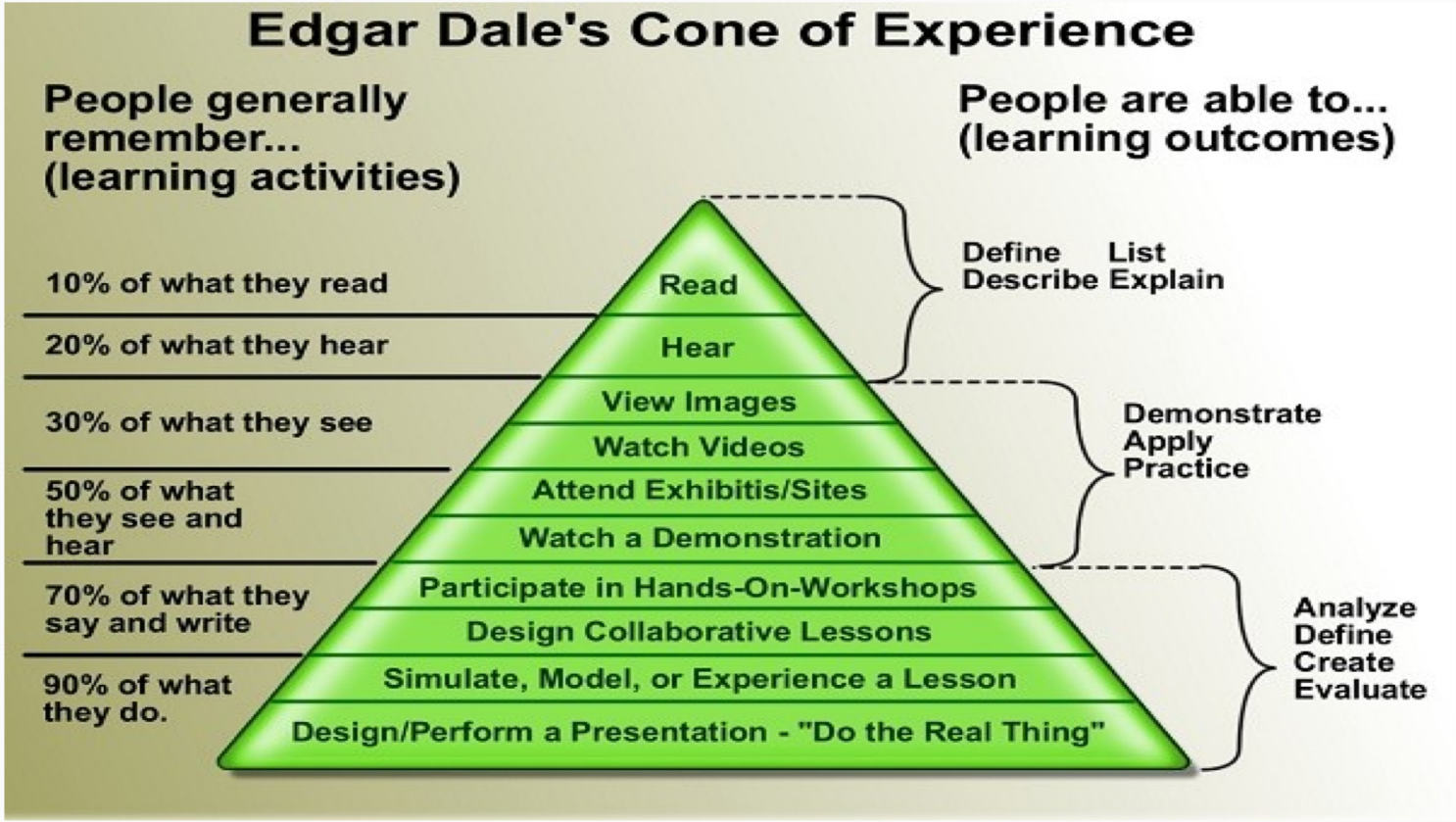
Agenda

1. Clean code / Clean architecture
2. MVVM using Kotlin
3. Dependency Injection using Dagger
4. NetworkBoundResources / MediatorLiveData
5. Data Binding
6. Google Architecture Components

Motivation

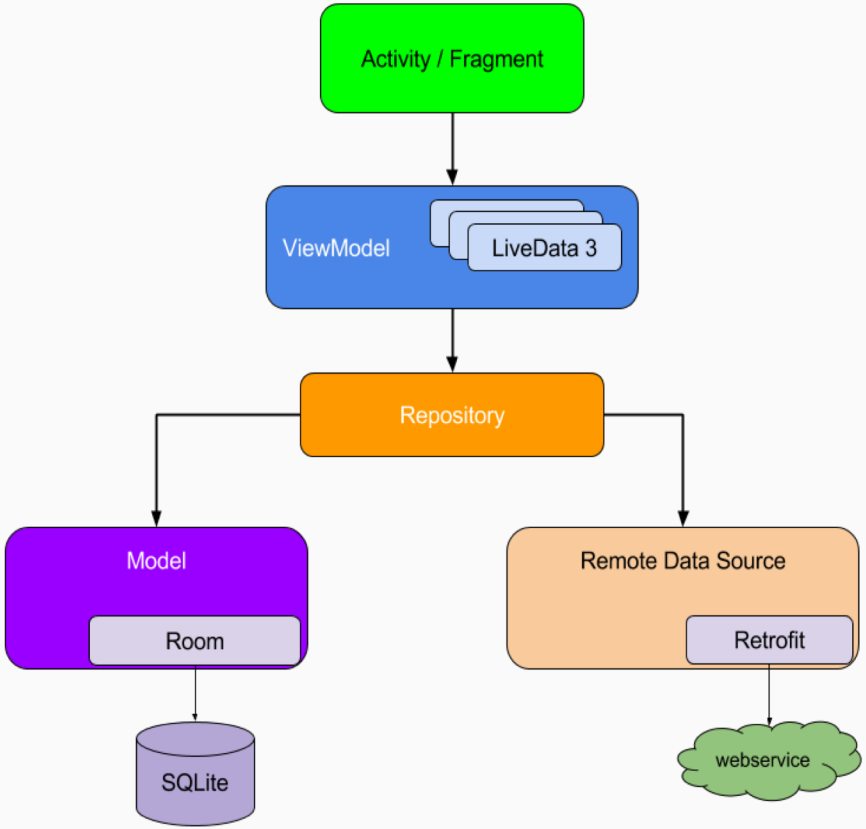
- Separation of concern (All functionalities/ Use cases are independent to each other)
- **Less bugs**
- No regression issues while adding or upgrading any features and UI
- Loose dependencies inside code
- **Less boilerplate** and easily maintainable code
- Transparent communication between all layers
- Streamlines the data sending process between layers
- Ease of unit testing
- **Ultimately reduced the development time** and efforts.

Learn efficiently

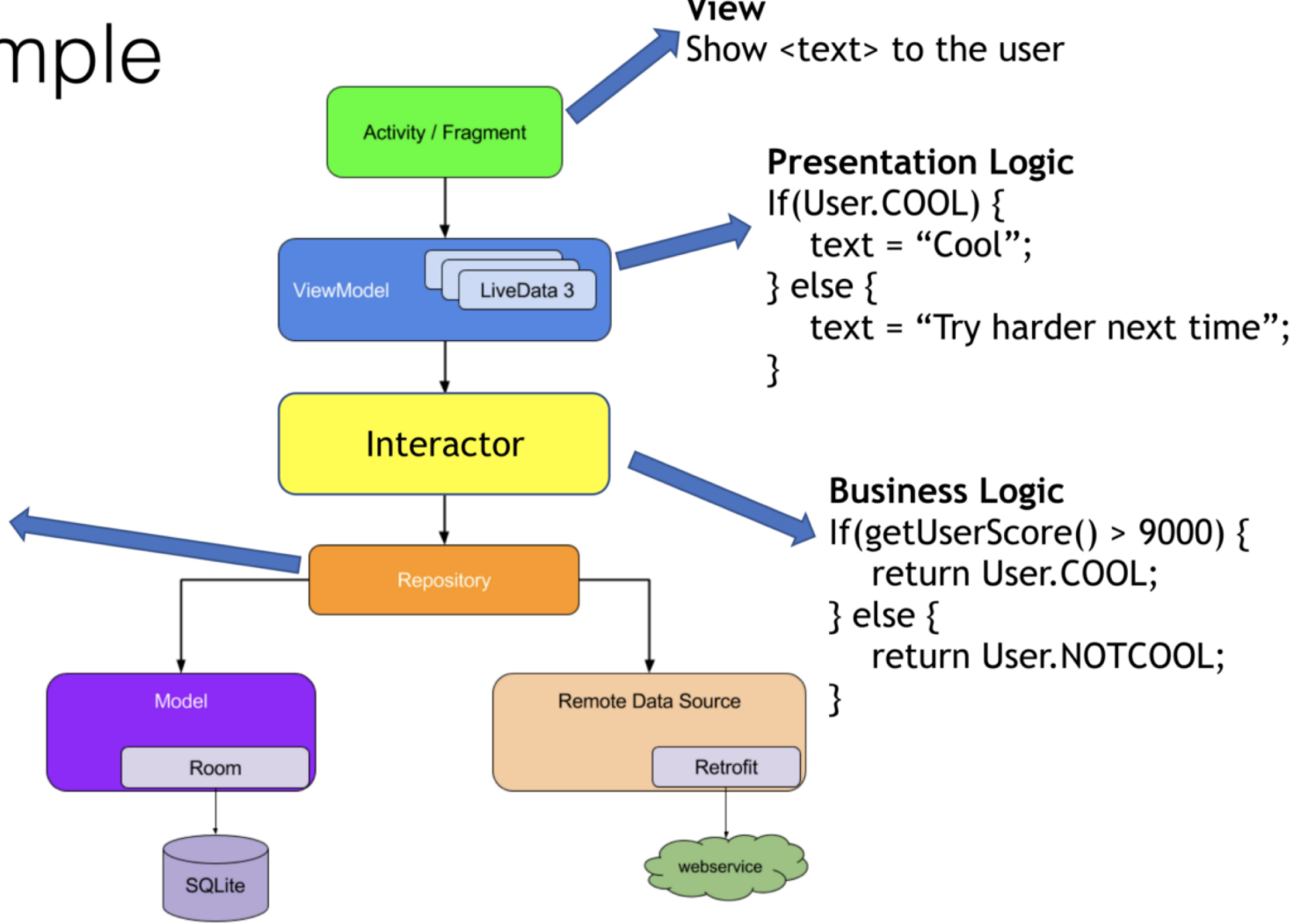


Why MVVM?

1. Official by android with android (jetpack)
2. Easy to separate and test all layers
3. **Less interfaces** not like MVP, less code
4. UI components are separated from business logic
5. Business logic is separated from Database
6. Easy to understand and read
7. Ensures your UI matches your data state and always up to date data
8. **No memory leaks**
9. No crashes due to stopped activities
10. **No more manual lifecycle handling**
11. Proper configuration changes
12. Sharing resources



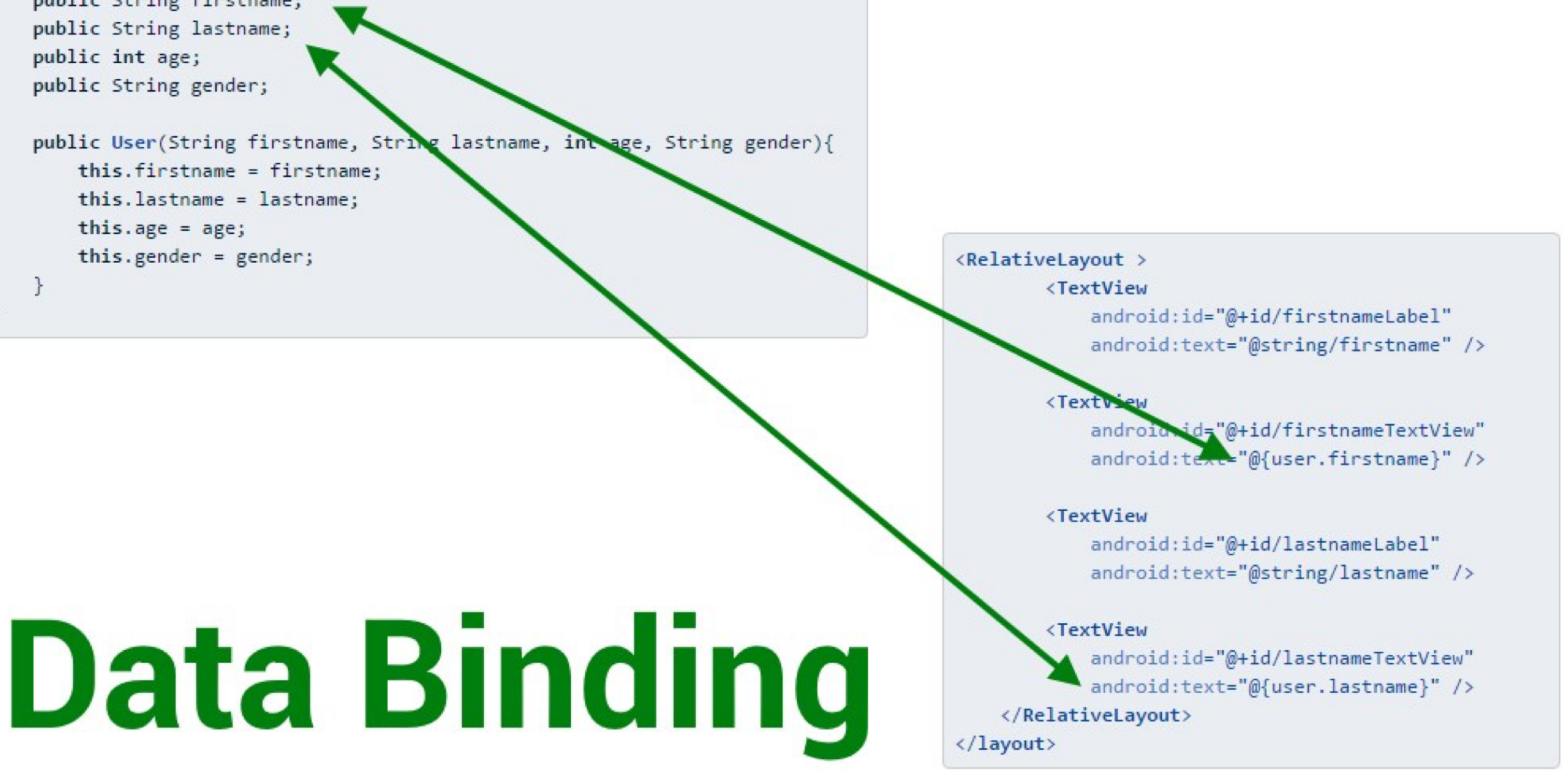
MVVMi simple example



Databinding

```
public class User {
    public String firstname;
    public String lastname;
    public int age;
    public String gender;

    public User(String firstname, String lastname, int age, String gender){
        this.firstname = firstname;
        this.lastname = lastname;
        this.age = age;
        this.gender = gender;
    }
}
```

```
<RelativeLayout >
        <TextView
            android:id="@+id/firstnameLabel"
            android:text="@string/firstname" />

        <TextView
            android:id="@+id/firstnameTextView"
            android:text="@{user.firstname}" />

        <TextView
            android:id="@+id/lastnameLabel"
            android:text="@string/lastname" />

        <TextView
            android:id="@+id/lastnameTextView"
            android:text="@{user.lastname}" />
    </RelativeLayout>
</layout>
```



Network Bound Resource using MediatorLiveData

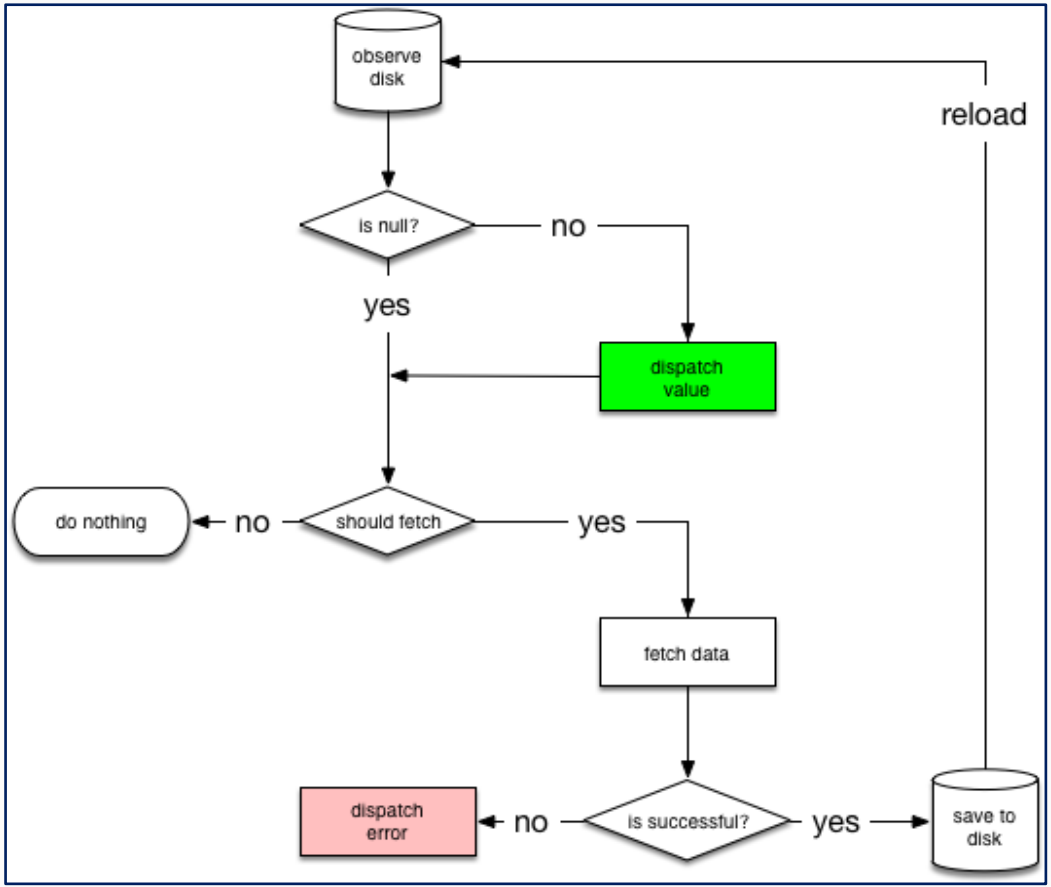
NetworkBoundResource is a helper class that use **MediatorLiveData**.

It's a common use case to load data from the network while showing the disk copy of the data for this situation can be easily handled with this helper class.

How NetworkBoundResource works?
Load data from the database for the first time, it checks whether the result is good enough to be dispatched or that it should be re-fetched from the network.

If the network call completes successfully, it saves the response into the database and re-initializes the stream.

https://developer.android.com/jetpack/docs/guide



NetworkBoundResource

```
abstract class NetworkAndDBBoundResource<ResultType, RequestType> @MainThread
            constructor(private val appExecutors: AppExecutors) {
    /**
     * The final result LiveData
     */
    private val result = MediatorLiveData<Resource<ResultType?>>()

    init {...}

    /**
     * Fetch the data from network and persist into DB and then
     * send it back to UI.
     */
    private fun fetchFromNetwork(dbSource: LiveData<ResultType>) {...}

    @MainThread
    private fun setValue(newValue: Resource<ResultType?>) {
        if (result.value != newValue) result.value = newValue
    }

    protected fun onFetchFailed() {}

    fun asLiveData(): LiveData<Resource<ResultType?>>
        return result
    }

    @WorkerThread
    private fun processResponse(response: Resource<RequestType>): RequestType? {
        return response.data
    }

    @WorkerThread
    protected abstract fun saveCallResult(item: RequestType)

    @MainThread
    protected abstract fun shouldFetch(data: ResultType?): Boolean

    @MainThread
    protected abstract fun loadFromDb(): LiveData<ResultType>

    @MainThread
    protected abstract fun createCall(): LiveData<Resource<RequestType>>
}
```

Manage to observe data from multiple sources e.g. Database, API, Cache etc.

A generic class that can provide a resource backed by both the SQLite database and the network.

Save response into database

Check your logic here , weather you want to fetch data from server or not, Internet check, cache expired etc.

Load already saved data from database

Create API call that points to your Retrofit API Call Service

```
/**
 * Fetch the data from network and persist into DB and then
 * send it back to UI.
 */
private fun fetchFromNetwork(dbSource: LiveData<ResultType>) {
    val apiResponse = createCall()
    // we re-attach dbSource as a new source, it will dispatch its latest value quickly
    result.addSource(dbSource) { it: ResultType!
        result.setValue(Resource.loading())
    }

    result.addSource(apiResponse) { response ->
        result.removeSource(dbSource)
        result.removeSource(apiResponse)

        response?.apply { this: Resource<RequestType>
            if (status.isSuccessful()) {
                appExecutors.diskIO().execute {

                    processResponse( response: this)?.let { requestType ->
                        saveCallResult(requestType)
                    }
                    appExecutors.mainThread().execute {
                        // we specially request a new live data,
                        // otherwise we will get immediately last cached value,
                        // which may not be updated with latest results received from network.
                        result.addSource(loadFromDb()) { newData ->
                            setValue(Resource.success(newData))
                        }
                    }
                }
            } else {
                onFetchFailed()
                result.addSource(dbSource) { it: ResultType!
                    result.setValue(Resource.error(errorMessage))
                }
            }
        }
    }
}
```

News Repository

```kotlin
@Singleton
class NewsRepository @Inject constructor(
    private val newsDao: NewsArticlesDao,
    private val apiServices: ApiServices, private val context: Context,
    private val appExecutors: AppExecutors = AppExecutors()
) {

    /**
     * Fetch the news articles from database if exist else fetch from web
     * and persist them in the database
     */
    fun getNewsArticles(countryShortKey: String): LiveData<Resource<List<NewsArticles>?>> {

        val data = HashMap<String, String>()
        data.put("country", countryShortKey)
        data.put("apiKey", BuildConfig.NEWS_API_KEY)

        return object : NetworkAndDBBoundResource<List<NewsArticles>, NewsSource>(appExecutors) {
            override fun saveCallResult(item: NewsSource) {
                if (!item.articles.isEmpty()) {
                    newsDao.deleteAllArticles()
                    newsDao.insertArticles(item.articles)
                }
            }

            override fun shouldFetch(data: List<NewsArticles>?) =
                (ConnectivityUtil.isConnected(context))

            override fun loadFromDb() = newsDao.getNewsArticles()

            override fun createCall() =
                apiServices.getNewsSource(data)

        }.asLiveData()
    }
```

Constructor injection for NewsDao, ApiService, Context, and AppExecutor using Dagger2

Resource? 🤔 *Don't worry it's on the way* 😅

Data object for request parameters

Save response into database

Check internet availability for should fetch implementation

Getting news articles from DB

```kotlin
/**
 * Fetch news articles from Google news using GET API Call on given Url
 * Url would be something like this top-headlines?country=my&apiKey=XYZ
 */
@GET("top-headlines")
fun getNewsSource(@QueryMap options: Map<String, String>): LiveData<Resource<NewsSource>>
```

Resource class: API status propagation

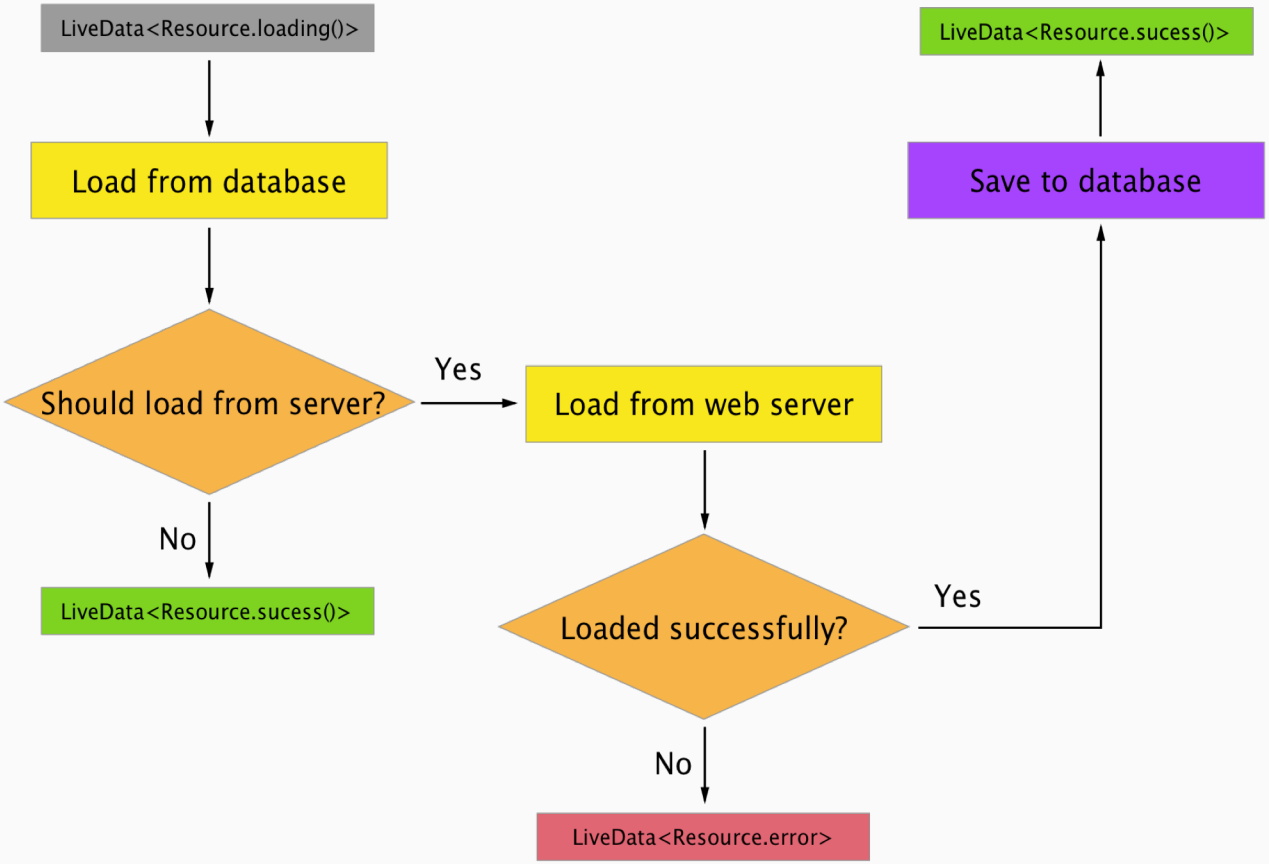
Resource class: A generic class that contains data and status about loading this data e.g.

1. **Resource.loading()**
2. **Resource.success()**
3. **Resource.error()**

MediatorLiveData:
LiveData subclass which may observe other LiveData objects and react on **OnChanged** events from them.

Scenario: We have 2 instances of LiveData, **liveData1** and **liveData2** , using **MediatorLiveData** we can merge their emissions in one LiveData Object.

https://developer.android.com/reference/androidx/lifecycle/MediatorLiveData.html



Resource class for API status

```kotlin
/**
 * A generic class that holds a value with its loading status.
 * @param <T>
</T> */
data class Resource<ResultType>(
    var status: Status,
    var data: ResultType? = null,
    var errorMessage: String? = null
) {

    companion object {
        /**
         * Creates [Resource] object with `SUCCESS` status and [data].
         * Returning object of Resource(Status.SUCCESS, data, null)
         * last value is null so passing it optionally
         *
         */
        fun <ResultType> success(data: ResultType): Resource<ResultType> =
            Resource(Status.SUCCESS, data)

        /**
         * Creates [Resource] object with `LOADING` status to notify
         * the UI to showing loading.
         * Returning object of Resource(Status.SUCCESS, null, null)
         * last two values are null so passing them optionally
         */
        fun <ResultType> loading(): Resource<ResultType> = Resource(Status.LOADING)

        /**
         * Creates [Resource] object with `ERROR` status and [message].
         * Returning object of Resource(Status.ERROR, errorMessage = message)
         */
        fun <ResultType> error(message: String?): Resource<ResultType> =
            Resource(Status.ERROR, errorMessage = message)

    }
}
```

Resource helper data class to share API response with proper data/status to UI.

In **kotlin** we declare static objects inside companion object block.

Success call-back to map success status with data.

Loading call-back to reflect loading status during an API Call.

Error call-back to send error status with proper error message.

ViewModel

Private LiveData field to hold reference from Repository as LiveData

Injecting repository to call API for data

Passing data received from UI and calling API from repository

This LiveData field would be observed in UI

```
/**
 * A container for [NewsArticles] related data to show on the UI.
 */
class NewsArticleViewModel @Inject constructor(
    private val newsRepository: NewsRepository
) : ViewModel() {

    /**
     * Loading news articles from internet and database
     */
    private fun newsArticles(countryKey: String): LiveData<Resource<List<NewsArticles>?>> =
        newsRepository.getNewsArticles(countryKey)

fun getNewsArticles(countryKey: String) = newsArticles(countryKey)
```

Observing Data in UI

```
/**
 * Get country news using Network & DB Bound Resource
 */
private fun getNewsOfCountry(countryKey: String) {
    /*
    * Observing for data change, Cater DB and Network Both
    * */
    newsArticleViewModel.getNewsArticles(countryKey).observe(this) {
        when {
            it.status.isLoading() -> {
                news_list.showProgressView()
            }
            it.status.isSuccessful() -> {
                it.load(news_list) {
                    // Update the UI as the data has changed
                    it?.let { adapter.replaceItems(it) }
                }
            }
            it.status.isError() -> {
                if (it.errorMessage != null)
                    ToastUtil.showCustomToast(this, it.errorMessage.toString())
            }
        }
    }
}
```

Observing news articles data from server

Loading call-back from Resource class

Success call-back you'll get your news list here

Error call-back

Depedency Injection with

Dagger 2 and Kotlin

Dependency Injection (DI) / Dagger2

1. Dagger2 is a Dependency Injection tool
2. Simple annotation structure
3. Readable generated code
4. No reflection (fast)
5. No runtime error
6. Pro-guard friendly
7. Written in pure Java
8. Annotations in code

https://dagger.dev/

Dependency Injection In Code

```
/* Without DI */
class CoffeeMaker {
  private final Heater heater;
  private final Pump pump;

  CoffeeMaker() {
    this.heater = new ElectricHeater();
    this.pump = new Thermosiphon(heater);
  }

  Coffee makeCoffee() {/* … */}
}

class CoffeeMain {
  public static void main(String[] args) {
    Coffee coffee =
      new CoffeeMaker().makeCoffee();
  }
}
```

Don't do this

```
/* With Manual DI */
class CoffeeMaker {
  private final Heater heater;
  private final Pump pump;

  CoffeeMaker(Heater heater, Pump pump) {
    this.heater = checkNotNull(heater);
    this.pump = checkNotNull(pump);
  }

  Coffee makeCoffee() {/* … */}
}

class CoffeeMain {
  public static void main(String[] args) {
    Heater heater = new ElectricHeater();
    Pump pump = new Thermosiphon(heater);
    Coffee coffee =
      new CoffeeMaker(heater, pump).makeCoffee();
  }
}
```

Do

Annotations in Dagger 2

1. **@Provides**
   Used on methods in classes annotated with **@Module** and is used for methods which provides objects for dependencies injection.

2. **@Module:** Used on classes which contains methods annotated with **@Provides**.

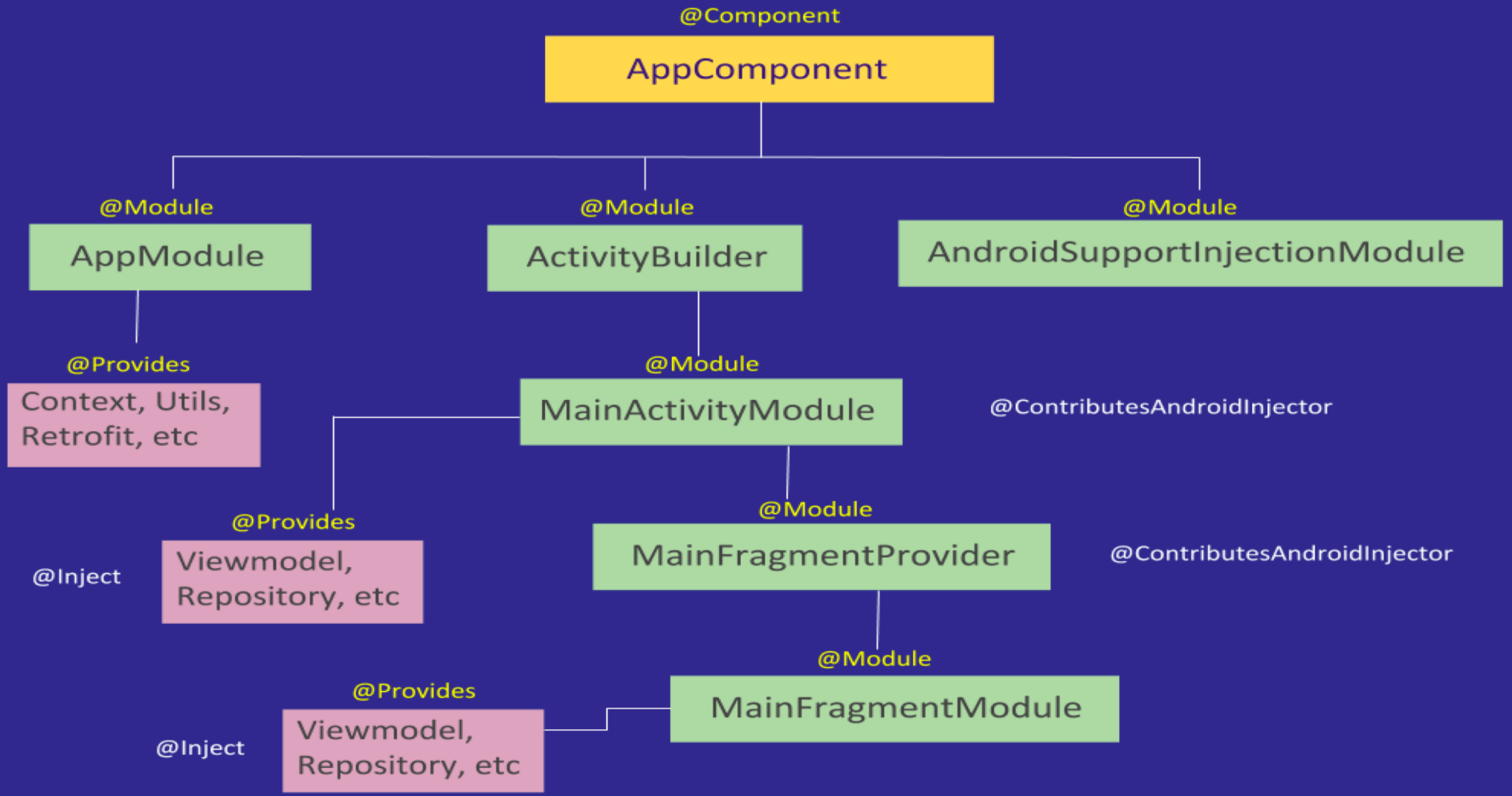
3. **@Component**
   Used on an interface. This interface is used by **Dagger2** to generate code which uses the modules to full fill the requested dependencies.

4. **@Inject (Inside Java):** Request dependencies. Can be used on a constructor, a field, or a method.

5. **@Qualifier (@Named):** Used to distinguish two different objects of the same return type.

6. **@Scope(@Singleton):** Single instance of this provided object is created and shared.

https://dagger.dev/


@Component
AppComponent
@Module
AppModule
@Module
ActivityBuilder
@Module
AndroidSupportInjectionModule
@Provides
Context, Utils, Retrofit, etc
@Module
MainActivityModule
@ContributesAndroidInjector
@Provides
Viewmodel, Repository, etc
@Inject
@Module
MainFragmentProvider
@ContributesAndroidInjector
@Module
MainFragmentModule
@Provides
Viewmodel, Repository, etc
@Inject

source

```kotlin
class App : Application(), HasActivityInjector {

    @Inject
    lateinit var dispatchingAndroidInjector: DispatchingAndroidInjector<Activity>

    override fun onCreate() {
        super.onCreate()

        AppInjector.init( app: this)
    }

    override fun activityInjector(): AndroidInjector<Activity> {
        return dispatchingAndroidInjector
    }
}
```

1. Application class

```kotlin
/**
 * Helper class to automatically inject fragments if they implement [Injectable].
 */
object AppInjector {

    fun init(app: App) {
        DaggerAppComponent.builder().application(app).build().inject(app)
```

2. AppInjector

```kotlin
/**
 * AndroidInjectionModule::class to support Dagger
 * AppModule::class is loading all modules for app
 */
@Singleton
@Component(modules = [AndroidInjectionModule::class, AppModule::class])
interface AppComponent {

    @Component.Builder
    interface Builder {

        @BindsInstance
        fun application(app: App): Builder

        fun build(): AppComponent
    }

    fun inject(app: App)
}
```

3. AppComponent

Refrence: Kotlin MVVM Architecture

```kotlin
@Module(includes = [PrefrencesModule::class, ActivityModule::class, ViewModelModule::class])
class AppModule {

    /**
     * Static variables to hold base url's etc.
     */
    companion object {
        private const val BASE_URL = BuildConfig.BASE_URL
    }


    /**
     * Provides ApiServices client for Retrofit
     */
    @Singleton
    @Provides
    fun provideNewsService(): ApiServices {
        return Retrofit.Builder()
            .baseUrl(BASE_URL)
            .addConverterFactory(GsonConverterFactory.create())
            .addCallAdapterFactory(LiveDataCallAdapterFactoryForRetrofit())
            .build()
            .create(ApiServices::class.java)
    }


    /**
     * Provides app AppDatabase
     */
    @Singleton
    @Provides
    fun provideDb(context: Context): AppDatabase {
        return Room.databaseBuilder(context, AppDatabase::class.java,  name: "news-db").build()
    }
```

4. AppModule

```kotlin
@Module
class PrefrencesModule {

    /**
     * Provides Preferences object with MODE_PRIVATE
     */
    @Provides
    @Singleton
    fun provideSharedPreference(app: Application): SharedPreferences =
        app.getSharedPreferences( name: "SharedPreferences", Context.MODE_PRIVATE)
```

5. PreferenceModule

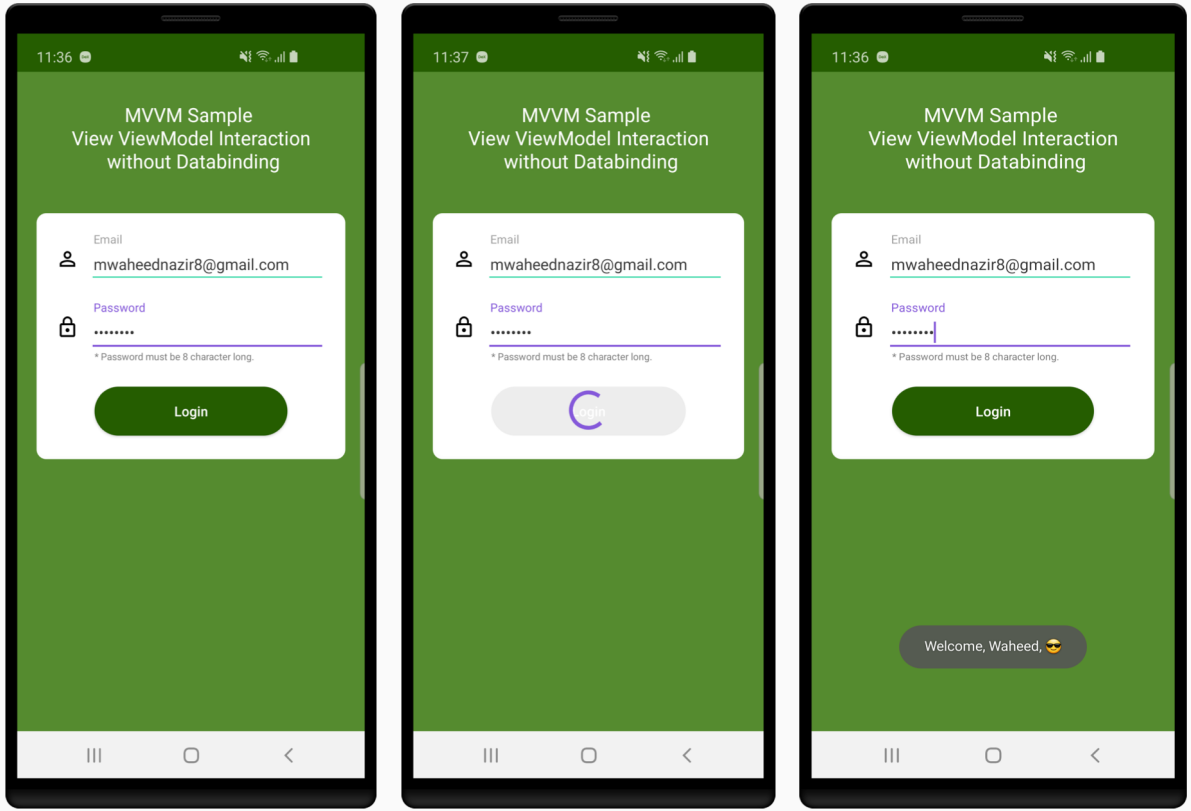
Demo: View – ViewModel – Communication

https://github.com/WaheedNazir/View-ViewModel-Interaction

MVVM, View and ViewModel Interaction using LiveData without using DataBinding in Kotlin.

Technology stack

1. MVVM
2. Live Data
3. Repository Pattern
4. AndroidX
5. JetPack Libraries



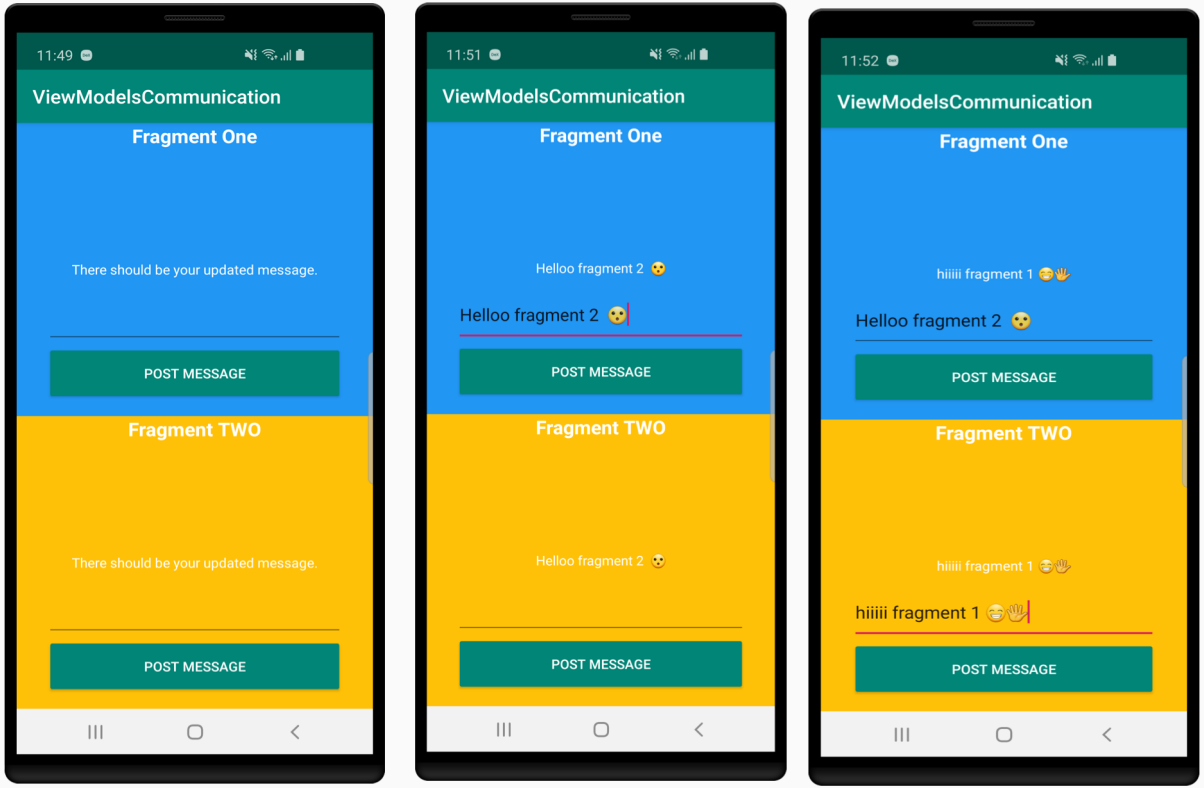
Demo: Communication – ActivityFragment – ViewModel

https://github.com/WaheedNazir/Communication-ActivityFragment-ViewModel

Shows communication between Activities/Fragments using ViewModels and LiveData in Kotlin.

Technology stack

1. MVVM
2. Live Data
3. Repository Pattern
4. AndroidX
5. JetPack Libraries



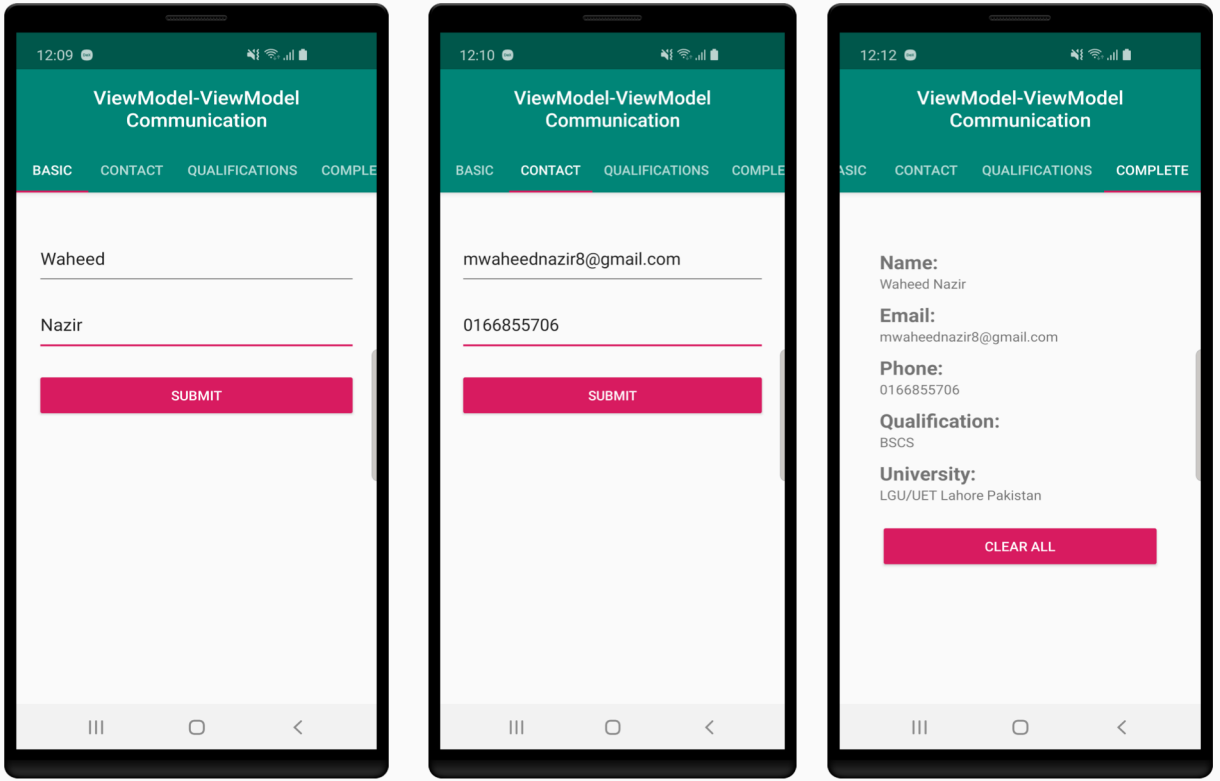
Demo: ViewModel – ViewModel – Communication

https://github.com/WaheedNazir/ViewModel-ViewModel-Communication

A sample app using Kotlin, MVVM, ViewModels, LiveData, display the communication between different ViewModels.

Technology stack

1. MVVM
2. Live Data
3. Repository Pattern
4. AndroidX
5. JetPack Libraries



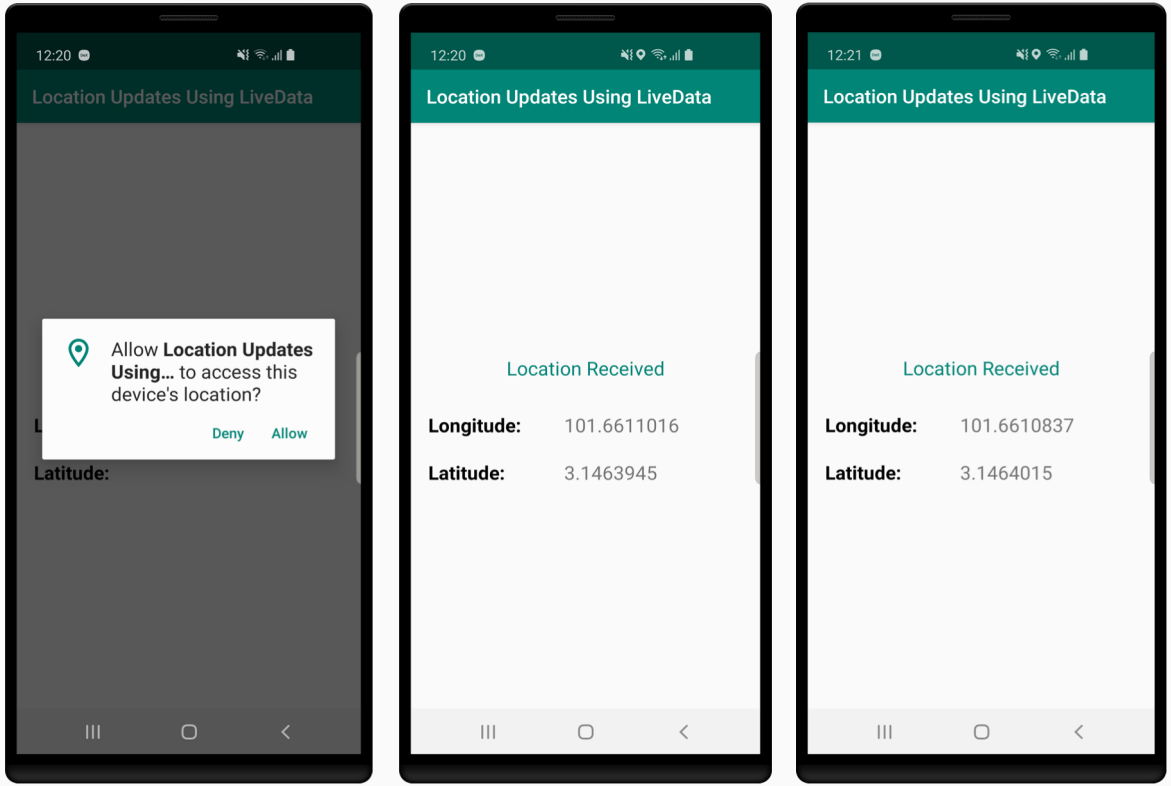
Demo: LocationUpdates-LiveData-ViewModel

https://github.com/WaheedNazir/LocationUpdates-LiveData-ViewModel

Location updates using LiveData, ViewModel, Fused Location API in Kotlin.

Technology stack

1. MVVM
2. Live Data
3. Repository Pattern
4. AndroidX
5. JetPack Libraries
6. FusedLocation API



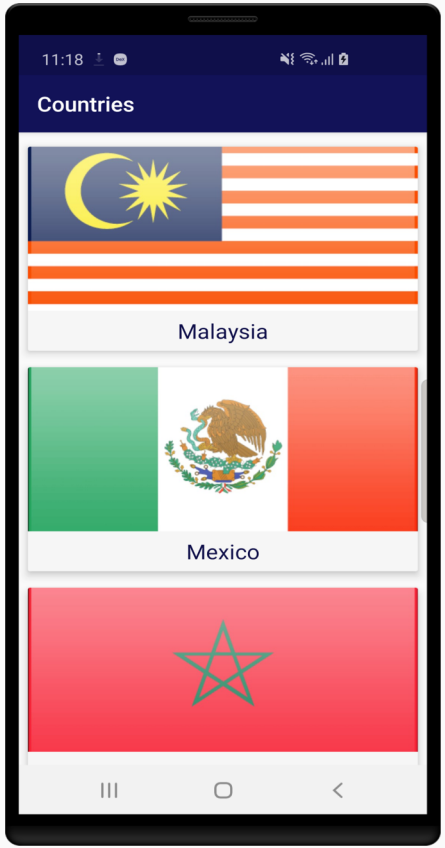
Demo / Bonus, Full Architecture

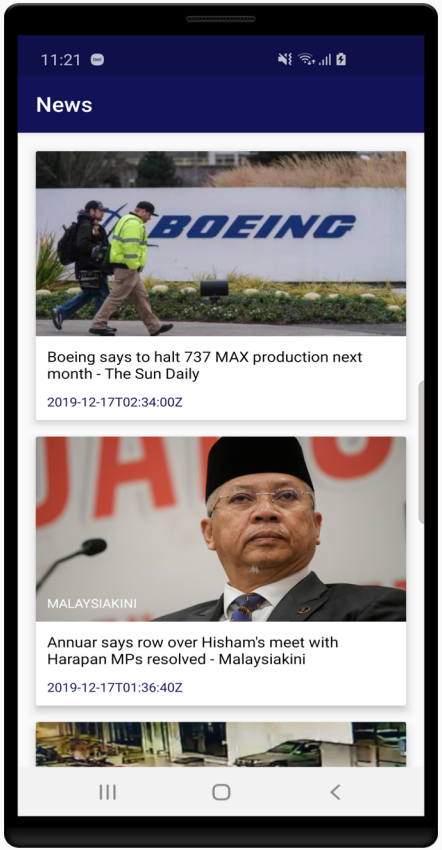
Kotlin MVVM Architecture

https://github.com/WaheedNazir/Kotlin-MVVM-Architecture

News listing app to show MVVM architecture design pattern using the following **technology stack.**

1. Architecture Design Pattern
2. MVVM
3. Dagger2 (Dependency Injection)
4. Live Data, MediatorLiveData
5. Room Database
6. Retrofit
7. Unit Testing (Espresso), Mockito (Coming soon)
8. Repository Pattern
9. AndroidX
10. Glide
11. NetworkBoundResource
12. JetPack Libraries





Important references

- Guide to app architecture android official by Google
- Lifecycle Aware Components by Google code lab
- Room, LiveData, and ViewModel by Google code lab
- Android Room with a View – Java by Google code lab
- Architecture Samples by Google
- LiveData with SnackBar, Navigation and other events (the SingleLiveEvent case)
- Unit-testing LiveData and other common observability problems
- Unlock your Android ViewModel power with Koin
- SOLID principles: Explanation and examples
- The Missing Google Sample of Android “Architecture Components” in Java Guide.
- Dagger 2 for Android Beginners the complete reference — Introduction in Java
- Dependency Injection with Dagger2, Detailed explanation
- Advanced Dagger 2 / Android and Kotlin
- Koin - Simple Android DI
- Dependency Injection With Koin
- Testing UI with Espresso
- Google Architecture Components