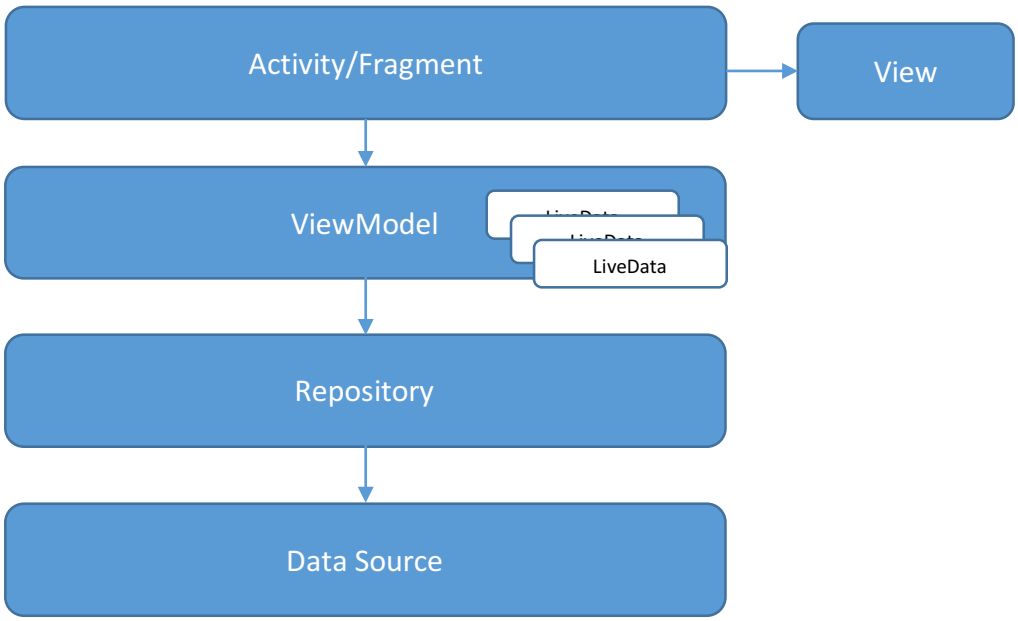
Applying Android Architecture Components with Kotlin

In this document we are going to look at the Android Architecture Components (AAC). We will do this by applying it on a demo App.

Architectural Pattern

Android Architecture Components (AAC) is a new collection of libraries that contains the lifecycle-aware components, which helps you design a robust, testable, and maintainable app.

For this demo we are combining the MVVM(Binder) pattern and the Android Architecture Components (ACC).



View: (Optional) The view is responsible for defining the structure, layout, and appearance of what the user sees on the screen. It uses Data binding to bind the data with the layout file. This part is extracted from the activity/fragment to make it easier to change the view or apply A/B testing. The view shouldn't hold any reference to the activity/fragment and it should expose the UI events (e.g. click events) through an event listener to the activity/fragment. You can also use Kotlin Android Extensions or Butter Knife.

Activity/Fragment: It is responsible for observing the ViewModel, keeping the UI up-to-date (through the View) and it also forwards the user actions back to the ViewModel. It shouldn't have any reference to layout file (e.g. by using findViewById), instead, the activity inflates the View as its content and implements view's event listeners.

ViewModel: It is responsible for preparing and keeping the data for the UI. It uses LiveData to keep the data and it survives the configuration changes. ViewModels are like gateway for the UI components. The UI components (Activity/Fragment) use ViewModels to communicate with the rest of the application.

LiveData: LiveData is a data holder class that keeps a value and allows this value to be observed. Unlike a regular observable (e.g. RxJava), LiveData respects the lifecycle of app components, such that the Observer can specify a Lifecycle in which it should observe.

Repository: Repository is responsible for the complete data model for the app, it provides simple data modification and retrieval APIs and it is responsible for fetching, syncing and persisting from different data sources.

Data Source: API to the data sources, such as Retrofit, Room and external contentProviders for the OS.

Demo App

Libs used in this app:

- Kotlin, the app is written in Kotlin programming language
- Dagger 2: to glue all parts together
- Retrofit: for REST API
- Glide: for the images
- Gson: JSON marshalling
- Android Studio 3.0 Canary 3.

This app is very simple. It has an input field where the user can input a search term. And it shows the top search result (a single product).

Components in the app:

- ProductDetailView (View)
- ProductDetailActivity (Activity)
- ProductViewModel (ViewModel)
- CatalogRepository (Repository)
- CatalogApi (Data source): the Retrofit interface

CatalogApi

A retrofit interface:

```
interface CatalogApi {

    @GET("catalog/v4/search")
    abstract fun search(@Query("q") q: String, @Query("limit") limit: Int = 10, @Query("offset") offset: Int = 0):
Call<SearchResponse>
}
```

CatalogRepository

```
class CatalogRepository @Inject constructor(val api: CatalogApi ) {

    fun doSearch(q: String, limit: Int = 1, offset: Int = 0): LiveData<Resource<SearchResponse>>{
        val data = MutableLiveData<Resource<SearchResponse>>();

        api.search(q, limit, offset).enqueue(object : Callback<SearchResponse> {
            override fun onResponse(call: Call<SearchResponse>?, response: Response<SearchResponse>?) {
                data.value = Resource.success(response?.body());
            }

            override fun onFailure(call: Call<SearchResponse>?, t: Throwable?) {
                val exception = AppException(t)
                data.value = Resource.error( exception)
            }
        });
        return data;
    }


}
```

The repository uses the CatalogApi to retrieve the data from the server. The CatalogApi is getting injected through the Dagger.

The doSearch function returns a *LiveData* of Resource of SearchResponse. We use the Resource class to hold the data and expose (network) errors.

```
class Resource<T> private constructor(val status: Resource.Status, val data: T?, val exception: AppException?) {
    enum class Status {
        SUCCESS, ERROR, LOADING
    }

    companion object {

        fun <T> success(data: T?): Resource<T> {
            return Resource(SUCCESS, data, null)
        }

        fun <T> error(exception: AppException?): Resource<T> {
            return Resource(ERROR, null, exception)
        }

        fun <T> loading(data: T?): Resource<T> {
            return Resource(LOADING, data, null)
        }
    }
}
```

ProductDetailActivity

Before we explain the ViewModel, let's look at the Activity and how it uses the ViewModel

```
class ProductDetailActivity : LifecycleActivity() {
    var productDetailViewModel: ProductViewModel? = null
    var view: ProductDetailView? = null

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        view = ProductDetailView(layoutInflater, object: ProductDetailView.Listener{
            override fun search(newValue: String) {
                productDetailViewModel?.search(newValue);
            }

        });
        setContentView(view?.rootView);

        productDetailViewModel = ProductViewModel.create(this);
        App.appComponent.inject(productDetailViewModel!!);

        productDetailViewModel?.searchResult?.observe(this, Observer<Resource<SearchResponse>> { resource ->
            if (resource != null) {
                when (resource.status) {
                    Resource.Status.SUCCESS -> {
                        val product = resource.data;
                        val products = product?.products;
                        if (products != null) {
                            if (!products.isEmpty()) {
                                view?.populateProduct(products.first())
                            }
                        }
                    }
                    Resource.Status.ERROR->{
                        Toast.makeText(this, "Error: "+resource.throwable?.message, Toast.LENGTH_LONG);
                    }
                }
            }
        })
    }
}
```

The ProductDetailActivity extends LifecycleActivity, that is because the Lifecycle implementation is not released yet, once it is released then the Activities and Fragments need to implement LifecycleOwner interface.

The activity holds a reference to *productDetailViewModel* which will be discussed soon. Once the productDetailViewModel is created, the activity starts *observing* the searchResult property of productDetailViewModel, which returns a *Resource<SearchResponse>* object. If the result is successful then it shows the product, otherwise it shows an error message.

The activity also creates a ProductDetailView object and implements the *ProductDetailView.Listener.search* method that returns the user input, which will be passed to productDetailViewModel.

ProductViewModel

```
class ProductViewModel: ViewModel(){
    val searchInput: MutableLiveData<String> = MutableLiveData()

    val searchResult = Transformations.switchMap(searchInput){
        if(it.length >= 1) {
            repository.doSearch(it)
        } else {
            MutableLiveData()
        }
    }

    private lateinit  var repository: CatalogRepository;

    @Inject fun init(repository: CatalogRepository) {
        this.repository = repository
```

```
    }

    fun search(term: String){
        searchInput.value = (term)
    }

    companion object{
        fun create(activity: FragmentActivity): ProductViewModel{
            var productDetailViewModel = ViewModelProviders.of(activity).get(ProductViewModel::class.java)
            return productDetailViewModel
        }
    }
}
```

ViewModelProviders is responsible for create and managing the lifecycle of ViewModels. We use a companion object (Java static method) method to return an instance of ProductDetailViewModel. If the activity is re-created (e.g. device rotated), it receives the same ProductDetailViewModel instance that was created before the recreation. When the owner activity is finished, the Framework calls ViewModel's onCleared() method so that it can clean up resources.

Since the consumer (in our case, the activity) is observing the *searchResult* attribute, we have to, somehow, propagate the active/inactive states down to it. For this purpose we can use *MediatorLiveData* or *Transformations.switchMap.* In this demo app, we use the later option.

The *Transformations.switchMap* function is used to set the value of the *searchResult*. This function is triggered by the *searchInput* LiveData object, which is set by the ProductDetailActivity (through *search* function). Transformations.switchMap creates a LiveData, *searchResult*, which follows the next flow: it reacts on changes of trigger LiveData (*searchInput*), applies the given function to new value of trigger LiveData and sets resulting LiveData as a "backing" LiveData to *searchResult*. "Backing" LiveData means, that all events emitted by it will retransmitted by *searchResult*.