Version 1.0

2018-JuL-22



Interview Feedback

Homework

Presenter: Arturo Sanchez

Mobile Apps Company

3301 Windy Ridge Parkway, Suite 400, Atlanta, GA 30339

# **Interview Feedback**

## Introduction

### Elevator

My name is Arturo Sanchez Chavarria, I am 28 years old and I have been working as an Analyst Developer since 2012.

I have 6 Years of I.T. experience – Software Development, 5 years as android development experience and 5 Android Apps in Play.

I have experience developing Android apps in versions of Android including Marshmallow, Lollipop, KitKat, Nougat, and Oreo.

A portfolio of published Android apps, enterprise and customer facing for notable, high profile companies – apps with high traffic.

### Previous Experience

I have experience developing mobile applications throughout the entire software development lifecycle (SDLC).

I have participated in all the phases including Requirements Collection, Design & Analysis of Customer Specifications, Development and Customization of the application, Unit testing and Production support, using Agile Scrum.

Skilled in Android development using Android SDK, frameworks, Java and Android Studio or Eclipse.

Experience in Messaging, Bluetooth and Wi-Fi connectivity with Android devices with Android CTS.

Experience on JavaScript and .NET client-server side development.

Experience with third-party APIs and web services like Google, Facebook, Twitter and YouTube Player.

Experienced integrating Android mobile apps with web services using Retrofit, Volley, AsyncTask, OkHTTP and JSON, GSON, REST.

Experience on every stage of application development right from collecting requirements, providing guidelines for design, creating application architecture, and implementation that includes various stages in development, QA, code signing and releasing to Market.

Worked with an internal QA team on system, performance, and acceptance testing.

## Questions Asked

### What are Broadcast receivers?

Is an Android component, which allows you to register for system or application events.

These components allow register for events that occur in the Android OS.

E.g.

* When a call is received,
* The battery is running low
* The smartphone is turned on
* An event is added to the calendar

### How to restrict a broadcast within the app?

Using the “Local Broadcast Manager”, for sending the broadcast.

The “Local Broadcast Manager” is a helper to register for and send broadcasts of Intents to local objects within your process.

Local Broadcast Manager Benefits:

1. Broadcast data won't leave your app, so don't need to worry about leaking private data.
2. It is not possible for other applications to send these broadcasts to your app, so you don't need to worry about having security holes they can exploit.
3. It is more efficient than sending a global broadcast through the system.

### What are StickyIntents and StickyBroadcast?

Sticky broadcast are broadcasts whose data is held by the system after being finished, so that clients can quickly retrieve that data without having to wait for the next broadcast.

Sticky Intent are for used for future broadcast listeners. For example if BATTERY\_LOW event occurs then that Intent will stick with Android so that any future requests for BATTERY\_LOW, will return the Intent.

### What design pattern does a broadcast receiver use?

Subscriber – publisher.

### Can we share data with another app using BR?

Yes, using the normal sendBroadcast method from our app, the other app can receive it if configured the broadcast receiver for the same action.

### How to secure a broadcast receiver?

1. Create the broadcast intent.
2. Declare the permission in the Android manifest using the <permission> element with the android:name attribute set to the name of the action.
3. Pass the Intent object and permission to the sendBroadcastWithPermission() method.
4. In the application that is supposed to receive this broadcast, you have to declare this permission and say that you use it.

### What are some ways to register a BR?

Static and Dynamic.

#### Broadcast Receiver - Static implementation

1. Create a class that extends from BroadcastReceiver

* Override the onReceive method

1. Add a receiver tag in the manifest

* name = ".YourReceiver"
* add the action you want to listen to inside the intent-filter

#### Broadcast Receiver - Dynamic implementation

1. Create a class that extends from BroadcastReceiver

* Override the onReceive method

1. On the component that you want to listen to the actions

* Override the "onStart" method
  + Create an instance of you receiver
  + Create an IntentFilter to pass an action (event(s) to register)
  + Call the Context.registerReceiver
* Override the "onStop" method
  + Call the Context.unregisterReceiver and pass the instance of your receiver as a parameter.

### What are some ways to make a network call?

* Native Call
* Using OkHttp
* Using Retrofit

### How can we make a network call on the main thread?

Using the Strict Mode Thread Policy.

### How to make a network call using Native way?

1. Create a Thread
2. Create an instance of a URL object passing the string Url as a parameter in the constructor.
3. Create an instance of the HttpURLConnection with the openConnection() method from the Url instance.
4. Call your request method (has several of them depending on the type of response you need).

### What is Retrofit? How to setup retrofit?

Is a type-safe REST client for Android developed by Square.

Turns your HTTP API into a Java interface.

Is the class through which your API interfaces are turned into callable objects. By default, Retrofit will give you sane defaults for your platform but it allows for customization.

#### Retrofit implementation

1. Add dependencies to the gradle

* Include the converter to be used.

1. Create Java Classes for Resources

* You can use a third-party tool to auto-generate these classes.

1. Creating the Retrofit instance

* To send out network requests to an API, you need to use the Retrofit builder class and specify the base URL for the service.
* You can also specify the converter factory to be used.

1. Define the Endpoints

* Endpoints are defined inside of an interface using special retrofit annotations to encode details about the parameters and request method.

1. Sending and receiving data

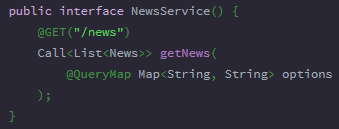
* Create methods to implement the Endpoints
* retrofit.create(Service.class)
* Create methods to consume the Call Responses.

### How to use Dynamic URLs in Retrofit?

1. In your Endpoint (inside of an interface using special retrofit annotations to encode details about the parameters and request method)
2. Define the annotation path with curly brackets ({}) for the variables to change.
   1. @GET(**"path/{dynamicpath}/restofthepath"**)  
      Observable<APIResponse> getInfo(@Path(**"dynamicpath"**) String dynamicpath);

### What if we have too many query params? How to add?

You can use the QueryMap annotation for the parameters and pass a Map object.



### What design pattern does RetroFit uses?

Façade Design patter.

Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.



#### Participants

The classes and objects participating in this pattern are:

* **Facade** 
  + knows which subsystem classes are responsible for a request.
  + delegates client requests to appropriate subsystem objects.
* **Subsystem classes** 
  + implement subsystem functionality.
  + handle work assigned by the Facade object.
  + have no knowledge of the facade and keep no reference to it.

### What are the benefits of RxJava?

* + 1. Simplifies the ability to chain async operations.
    2. Exposes a more explicit way for declaring how concurrent operations should operate.
    3. Surfaces errors sooner.
    4. Helps reduce the need for state variables that can introduce bugs.

### What are some types of Observables?

1. Observable <> Observer

* This is the simplest Observable which can emit more than one value.

1. Flowable <> Observer

* When there is a case that the Observable is emitting huge numbers of values which can't be consumed by the Observer.
* The Flowable Observable handles the exception with a strategy.

1. Single <> Single Observer

* Single is used when the Observable has to emit only one value like a response from a network call.

1. Maybe <> Maybe Observer

* Maybe is used when the Observable has to emit a value or no value.

1. Completable <> Completable Observer

* Completable is used when the Observable has to do some task without emitting a value.

### What is difference between Map and FlatMap?

Map transform one event to another. FlatMap transform one event to zero or more event.

FlatMap behaves very much like map, the difference is that the function it applies returns an observable itself, so it's perfectly suited to map over asynchronous operations.

In the practical sense, the function Map applies just makes a transformation over the chained response (not returning an Observable); while the function FlatMap applies returns an Observable<T>, that is why FlatMap is recommended if you plan to make an asynchronous call inside the method.

Summary:

Map returns an object of type T

FlatMap returns an Observable.

### What is Dagger? Why DI? Way to inject?

The general concept behind dependency injection is called Inversion of Control.

According to this concept, a class should not configure its dependencies statically but should be configured from the outside.

Dagger is a fully static, compile-time dependency injection framework for both Java and Android.

It makes it easier to manage the dependencies between the classes in our app.

#### Advantages of Dependency Injection

1. Reduced Dependencies
2. Reduced Dependency Carrying
3. More Reusable Code
4. More Testable Code
5. More Readable Code

#### Ways to inject your dependencies into your class

* constructor
* method
* field injection

### What is the difference between Dagger 1 and Dagger 2?

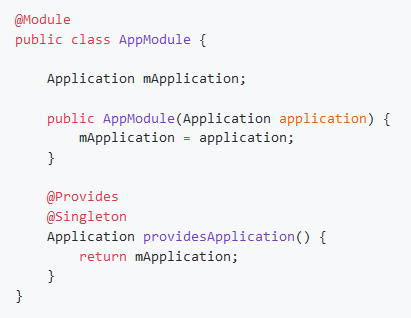
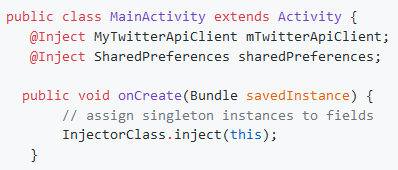
Advantages of Dagger 2:

* No more reflection - everything is done as concrete calls
* No more runtime graph composition - improves performance, including the per-request cases
* Traceable - better generated code and no reflection help make the code readable and easy to follow
* Supports method injection in addition to field and constructor injection which were the only two types supported by Dagger 1
* Modules require less configuration than Dagger 1
* Allows users to use any well-formed scope annotation. Dagger 1 only supported a single scope: @Singleton.

And disadvantages:

* The inject() method now has a strong type association with the injection target. This is good for debugging, but it complicates a common practice of injecting from base classes (e.g. base activities, fragments etc).
* Component implementation requires rebuilding the project to appear and any injection-related compile errors result in the class disappearing (i.e. not being generated).
* Doesn't support overrides. Modules that override for simple testing fakes can create a subclass of the module to emulate that behavior. Modules that use overrides and rely on dependency injection should be decomposed so that the overriden modules are instead represented as a choice between two modules.

### How do we setup Dagger? Explain each component?

1. Add dependencies to Gradle
2. Declare your singletons
   1. 
3. Define injection targets
   1. 
4. Code generation
   1. An important aspect of Dagger 2 is that the library generates code for classes annotated with the @Component interface. You can use a class prefixed with Dagger (i.e. DaggerTwitterApiComponent.java) that will be responsible for instantiating an instance of our dependency graph and using it to perform the injection work for fields annotated with @Inject.
5. Instantiating the component
   1. Make sure to rebuild the project (in Android Studio, select Build > Rebuild Project) if you cannot reference the Dagger component.

### MVP? Difference between MVC and MVP?

#### MVC

The model, view, controller approach separates your application at a macro level into 3 sets of responsibilities.

##### Model

The model is the **Data + State + Business logic** our application. It is the brains of our application so to speak. It is not tied to the view or controller, and because of this, it is reusable in many contexts.

##### View

The view is the **Representation** of the Model. The view has a responsibility to render the User Interface (UI) and communicate to the controller when the user interacts with the application. In MVC architecture, Views are generally pretty “dumb” in that they have no knowledge of the underlying model and no understanding of state or what to do when a user interacts by clicking a button, typing a value, etc. The idea is that the less they know the more loosely coupled they are to the model and therefore the more flexible they are to change.

##### Controller

The controller is **Glue** that ties the app together. It is the master controller for what happens in the application. When the View tells the controller that a user clicked a button, the controller decides how to interact with the model accordingly. Based on data changing in the model, the controller may decide to update the state of the view as appropriate. In the case of an Android application, the controller is usually represented by an Activity or Fragment.



#### Evaluation

MVC does a great job of separating the model and view. Certainly the model can be easily tested because it is not tied to anything and the view has nothing much to test at a unit testing level. The Controller has a few problems however.

##### Controller Concerns

* Testability - The controller is tied so tightly to the Android APIs that it is difficult to unit test.
* Modularity & Flexibility - The controllers are tightly coupled to the views. It might as well be an extension of the view. If we change the view, we have to go back and change the controller.
* Maintenance - Over time, particularly in applications with anemic models, more and more code starts getting transferred into the controllers, making them bloated and brittle.

#### MVP

MVP breaks the controller up so that the natural view/activity coupling can occur without tying it to the rest of the “controller” responsibilities.

##### Model

Same as MVC / No change.

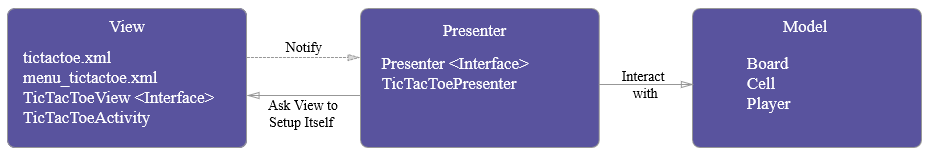
The model is the **Data + State + Business logic** our application. It is the brains of our application so to speak. It is not tied to the view or controller, and because of this, it is reusable in many contexts.

##### View

The only change here is that the Activity/Fragment is now considered part of the view. We stop fighting the natural tendency for them to go hand in hand. Good practice is to have the Activity implement a view interface so that the presenter has an interface to code to. This eliminates coupling it to any specific view and allows simple unit testing with a mock implementation of the view.

##### Presenter

This is essentially the controller from MVC except that it is not at all tied to the View, just an interface. This addresses the testability concerns as well as the modularity/flexibility concerns we had with MVC. In fact, MVP purists would argue that the presenter should never have any references to any Android APIs or code.



#### Evaluation

This is much cleaner. We can easily unit test the presenter logic because it’s not tied to any Android specific views and APIs and that also allows us to work with any other view as long as the view implements the ***View*** interface.

##### Presenter Concerns

* Maintenance - Presenters, just like Controllers, are prone to collecting additional business logic, sprinkled in, over time. At some point, developers often find themselves with large unwieldy presenters that are difficult to break apart.

Of course, the careful developer can help to prevent this, by diligently guarding against this temptation as the application changes over time.

### How do we call the SharedPreferences using MVP?

1. Create a Model Class for the preferences in the APP
2. Create a contract interface for the presenter
   1. It will have a signature method for showing the preferences.
3. Create a Presenter that extends from the contract and gets a Shared Preferences Instance
   1. Override the method for showing the preferences.
      1. Mapping from the Shared Preferences Instance to the Model.
      2. Returns the Model
4. In the activity (or other component) that will use the result from the Presenter
   1. Implement the contract interface for the view
   2. Create an instance for the presenter
   3. Attach the view to the presenter.
   4. Call the Method for the loading of the shared preferences.
      1. Pass the Shared Preferences Instance as a parameter.
   5. Override the method for showing the shared preferences
      1. Handle the result.