Architecture decisions
In this project, I used 2 major architecture patterns: Clean architecture (by Uncle Bob) and MVP
Clean architecture:
http://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html
MVP: Model-View-Presenter pattern

With those patterns it is easy to define the application layers and responsibilities. The app was split into 3 modules: Domain, Data and App. To achieve independence between these modules, each layer has its own models and converters.

The **domain layer** contains all the business rules, use cases and application models. This is a pure Kotlin module.
I also used this layer as a thread switch. The use cases receive their requests in the main thread, switch to a background thread and return their results back to the main thread. This way I do not have to be extra careful on running network request or making DB queries in the main thread. To achieve this I used Coroutines.

The **data layer** is responsible for providing data to the Domain layer. In this layer is widely used the dependency inversion pattern, as we have network dependency, I have chosen to use a memory cache here as well, but it can easily be replaced by any other cache strategy (DB, Files, etc).

The **app layer** is where all the Android framework dependencies are implemented. I decided to use Koin as dependency injector. This layer also contains the MVP implementation, which was chosen as it is easy the view logic (which must be dummy) by the presenter. This is the only layer with Android dependency.

Libraries
It was used Retrofit and OkHttp to make the network requests;
Koin to dependency injection;
Picasso for image downloading;
It was also added some libraries to help the development and debug: Stetho and LeakCanary;
For tests purposes, it was basically used Mockito, JUnit and MockWebServer;

Testing
To develop this app it was used mainly TDD. All the layers were unit tested. The UI tests were not implemented, the views were tested by their attached presenter. This decision was taken mainly due to time.

Improvements / technical debts
Add instrumentation tests: We can develop these tests using Espresso.
Add another layer for interaction between presenter and use-cases. When the presenter starts to grow too much we can use this layer to keep it simpler.
Clean resource files creating styles for texts and common components.