**Design Patterns**

Centauri

**Repository**

The project makes heavy use of repositories to abstract away data persistence. The repository facilitates the mediation between the ViewModel + Domain layer to the Data layer, both by transforming the data into their Data/Domain counterparts respectively and by facilitating where the data comes from in an ordered hierarchy.

The hierarchy is a standard hierarchy that mimics most industry apps, with the first reference being from the cache (HTTP GET request cache, Data Cache in project, or SharedPreferences).



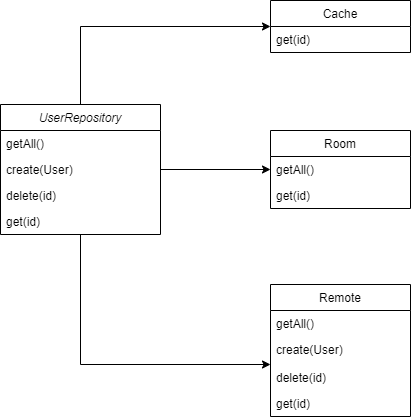
The cache object will make requests to whatever is suitable for the object it is requesting. If it is requesting a user session, it will request the SharedPreferences, as the sessions are mapped by the userid in that.

The second request the data will make if the former does not exist is it will request the Room database, as discussed briefly in the Architecture document. Data gets routinely stored in the Room database after requests are made to the server. In the case that data gets updated on the server and the client doesn’t know about it, the data on the Room database gets invalidated after 24 hours. If the client doesn’t want to wait for this, the cache can get cleared to have changes from the remote API be reflected in the client.

The third and final request will be to the remote API. The remote API is facilitated by Retrofit2.0 and RxJava2, which create asynchronous HTTPS(TLS 1.2) requests to the server. Since this is the longest request, this is the last in the hierarchy, and is only requested when neither of the two exist.

In total, the repository is the core of our project, and abstracts all of the data persistence in our project, along with comprising itself of multiple functions that allow for simple requests to be made in the Domain/Business layer.

Small UML Diagram of the base representation of the Repository.



In this case, it shows the bare Repository of the User model. In the actual implementation, the Repository also handles requests such as logging in, typically left to objects like the DAO. In this case, the Repository can be considered akin to the DAO design pattern.

**Observer/Publish+Subscribe**

Since our app runs off of the MVVM architecture as discussed in the Software Architecture document, the main way that our UI updates is through the Observer + Subscriber pattern. In our case, objects that are being received asynchronously are declared as MutableLiveData<ThatObjectType>. When doing so, we can call setValue(newValue) to update the objects value, which then will be observed in the View of the app.

The main process is reflected here:



In total, the View will make requests for data on user actions (opening the View, clicking buttons, entering in EditTexts, etc.) and the ViewModel will forward the request to the Repository. The request will be asynchronous, so the View cannot expect an update immediately. Due to this fact, we must use observers and RxJava2, which allows us to observe the repository request. When the request is received or it errors out, the LiveData will be updated accordingly. In doing so, the View’s observers on the LiveData will thus update the View for the user, or show an error message indicating what went wrong.

**Singleton**

The Singleton pattern is a slightly less important design pattern in our product, mainly due to its replaceability with a design that doesn’t include the Singleton. In our case, the Repository acts as a Singleton for our application, mainly due to the fact that there doesn’t need to exist another reference to the Repository(ies).

In our case, the ViewModels are the only classes that hold references to the Repository. Thus the interaction is shown as follows:

